# CAUSE-EFFECT GRAPHING

# USER GUIDE

**Technical Support**
If you have any questions about this manual or this software, contact the Bender RBT Inc. support group. The hours are Monday-Friday, 8:00 AM to 6:00 PM PST. The fax number and e-mail options are available 24 hours a day, 7 days a week.

- E-mail: support@BenderRBT.com
- Phone: (707) 538-1932
- Fax: (707) 538-1481

**Table of Contents**

# 1. Introduction

There are two challenges in designing a good set of tests for software: 1. you need to minimize the number of tests while still providing strong coverage and 2. you need to ensure that you are getting the right answer for the right reason.

For even a relatively simple system the number of possible test suites actually exceeds the number of molecules in the universe (which is $10^{80}$ according to Stephen Hawking in "A Brief History In Time"). The key challenge then is to select an infinitesimally small subset of tests which, if they run correctly, give you a very high degree of assurance that all of the other combinations/permutations will also run correctly.

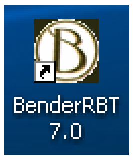
The issue in ensuring that you got the right answer for the right reason involves the fact that two or more defects may cancel each other out under some circumstances. You get the right answer for the wrong reason. To solve this, tests must “sensitized” to ensure any defects will be seen at an observable point.

The Cause-Effect Graphing test design engine portion of BenderRBT addresses both test optimization and observability of defects since it sensitizes the test paths to ensure that any logic defect will propagate to an observable point. If the functions you are testing are business critical, mission critical, and/or safety critical, we recommend using the Cause-Effect Graph based test design. The other test design engine in BenderRBT is Quick Design. It is based on pair-wise testing. It and all other combinatorics based test design engines address only reducing the number of tests to a manageable level. Quick Design should be used for non-critical functions or an initial shakedown of critical functions – which would then be followed by C-E Graph based tests. Another area where Quick Design is appropriate is in designing configuration tests and creating seed tests for performance testing. (See the Quick Design User Manual for the details of using that test design engine.)

Tools automate a process. Actually, automation without good process is doomed to fail. When we teach the Requirements Based Testing (RBT) class, 95% of the time is spent on process, not the tool. This user manual will assume that you are already familiar with the overall RBT process and Cause-Effect Graphing specifically. If that is not true, then we suggest that you read the RBT Process Tutorial which is included in the Documentation directory and/or take the RBT class.

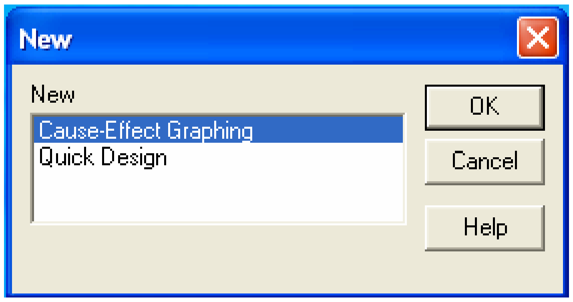
# 2. Entering Cause-Effect Graphing

BenderRBT includes two test design engines. If you double click on the BenderRBT icon



**RBT Icon**

you will be presented with a choice



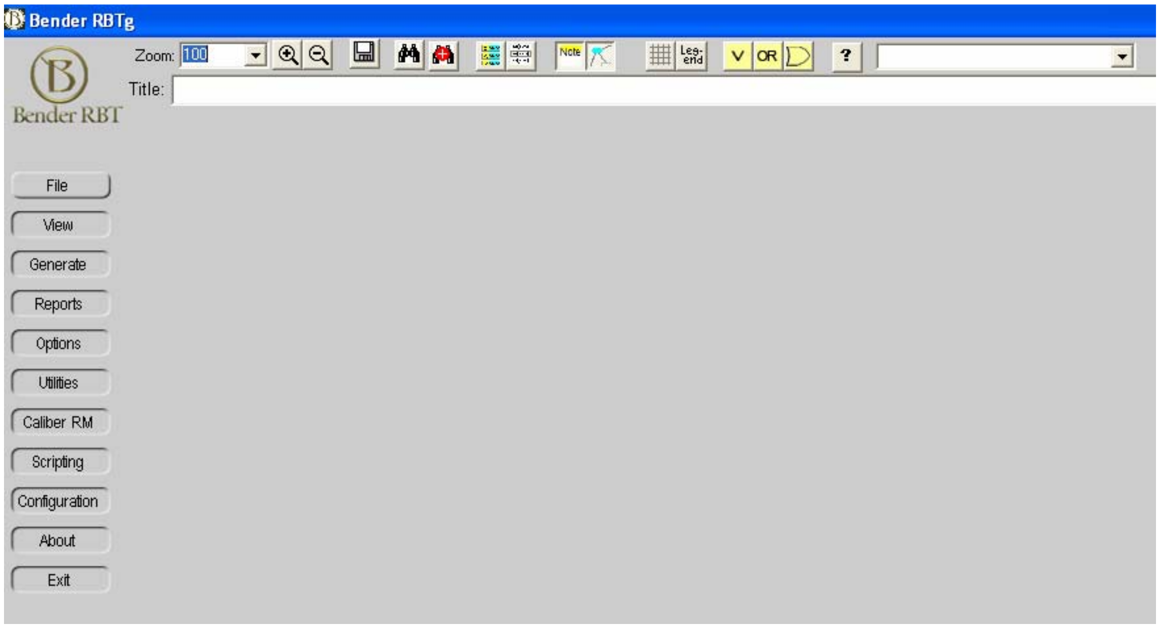
**RBT Test Design Engine Options**

of Cause-Effect Graphing or Quick Design. Quick Design takes you to the main screen of the pairs-wised based test design engine. To enter Cause-Effect Graphing, select it and hit OK. This will take you to the graphing component.

The Cause-Effect Graphing component is actually composed of two components – RBTg and RBT. RBTg is the graphic front end. You draw your graphs in this component. RBTg was developed for us by Software Prototype Technologies Inc. (SPT). RBTg then translates the graphic model into the API format used by the RBT test case design engine. Before the graphic front end - either the current one or the earlier Visio based one - the input to the earliest versions of RBT (then called SoftTest and later Caliber-RBT) was via a character interface based on Prolog syntax. This interface still exists. In addition to RBTg, it also can be used by other tools to directly generate the graph input from a requirement without a person having to draw the graph. For example, we are currently working with Unisys to create a link between their Rules Modeler (RM) product and RBT. An analyst would define the requirements in RM. RM would then export selected information to RBT via the API to generate the tests automatically. RBT would also validate the logical consistency of the requirements.

A full description of the API can be found in Appendix A.

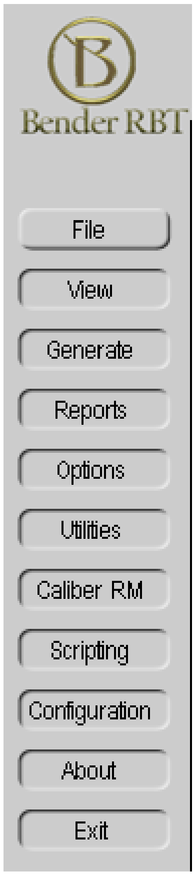
On initially entering you will get the main Cause-Effect Graphing screen:



**C-E Graphing Start Up Screen**

You are now ready to get started. In the following sections we will go step by step in creating a graph, generating the tests, reviewing the reports, using various utilities, and exporting the results to other tools.

The main menu of RBTg consists of a number of options:



The **File** menu addresses classic open, close, save, and save as functions.

The **View** menu addresses what to display in the work area.

The **Generate** menu addresses various options for creating, evaluating, and updating your test cases.

The **Reports, Options**, and **Utilities** menus take you directly to the RBT engine where these features are available.

The **Caliber RM** option takes you to RBT and ensures that the link to RM is set up properly.

The **Scripting** option is only active if you are using the Direct To Test (DTT) version of RBT. This is used to generate automated scripts for various playback tools.

The **Configuration** option allows you to access RBT when it is installed in other than the default path.

The **About** and **Exit** options mean what they do in all applications.

# 3. Creating the Cause-Effect Graph

For the following discussion on creating a graph and reviewing the functions/features of the tool we will use this specification:

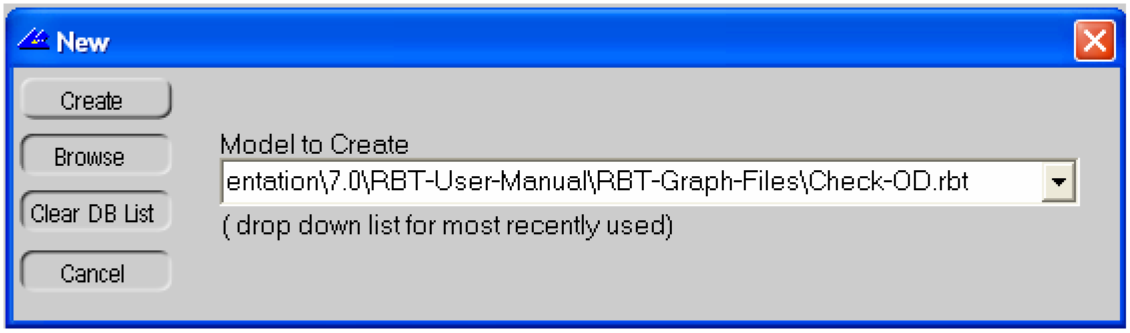
> **Overdraft Protection Specification**
> If the customer is a business client or a preferred personal client and they have a checking account, $100,000 or more in deposits, no overdraft protection and fewer than 5 overdrafts in the last 12 months, set up free overdraft protection. Else, do not give them free overdraft protection.

The corresponding RBT file is called Checkod.rbt and can be found in the Examples directory where RBT was installed.

## 3.1 *Creating a New RBT File*

The Cause-Effect Graphs are created in an .rbt file. This file contains information about the nodes, relations, constraints, etc. When you run this file to generate the tests two other files will be created: the .ceg file (which is the API data file) and the d_b file (which contains all of the generated test case definition data). These are work files. They are given the same name as the .rbt file and placed in the same directory.

To create a new graph, select **File → New**. The following dialog will appear:



Model to Create
entation\7.0\RBT-User-Manual\RBT-Graph-Files\Check-OD.rbt
( drop down list for most recently used)

**Create New C-E Graph File**

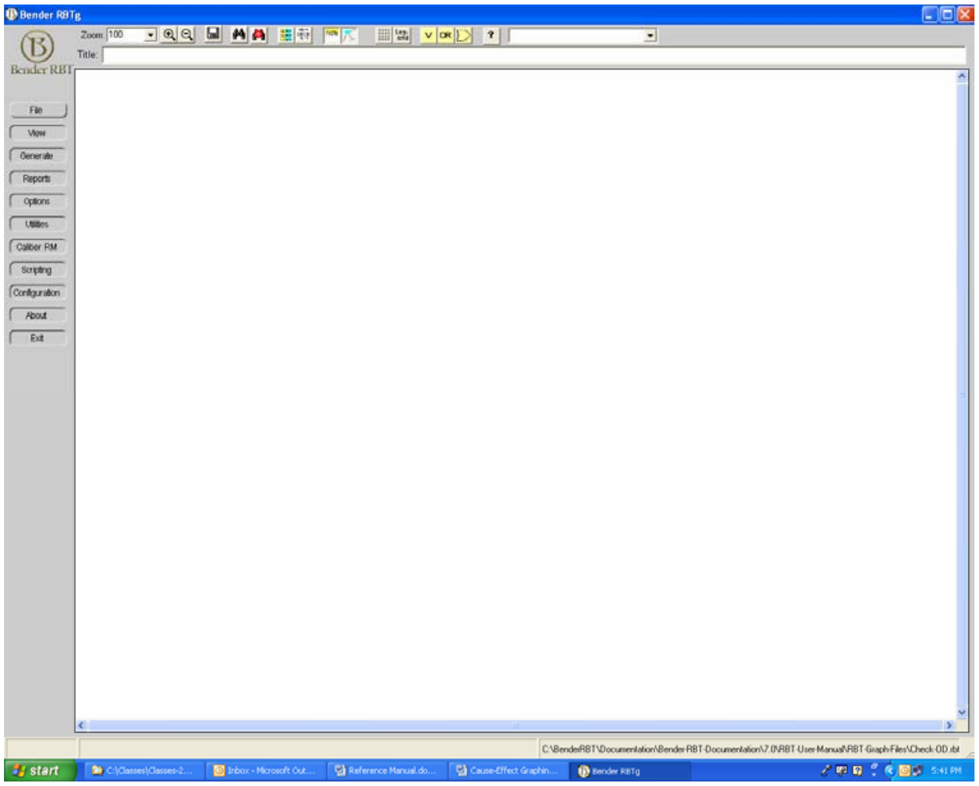
The "Browse" option allows you to select the directory in which the new graph should be placed.

The "Clear DB List" button will blank out the contents of the file name window and the most recently used list.

The pull down will show you the list of most recently used graphs.

In this case we will name our graph file Check-OD. You then hit the "Create" button. RBT will create a new data base for this graph. (It is actually an Access data base.)

This brings up the screen where you can now begin drawing your graph:

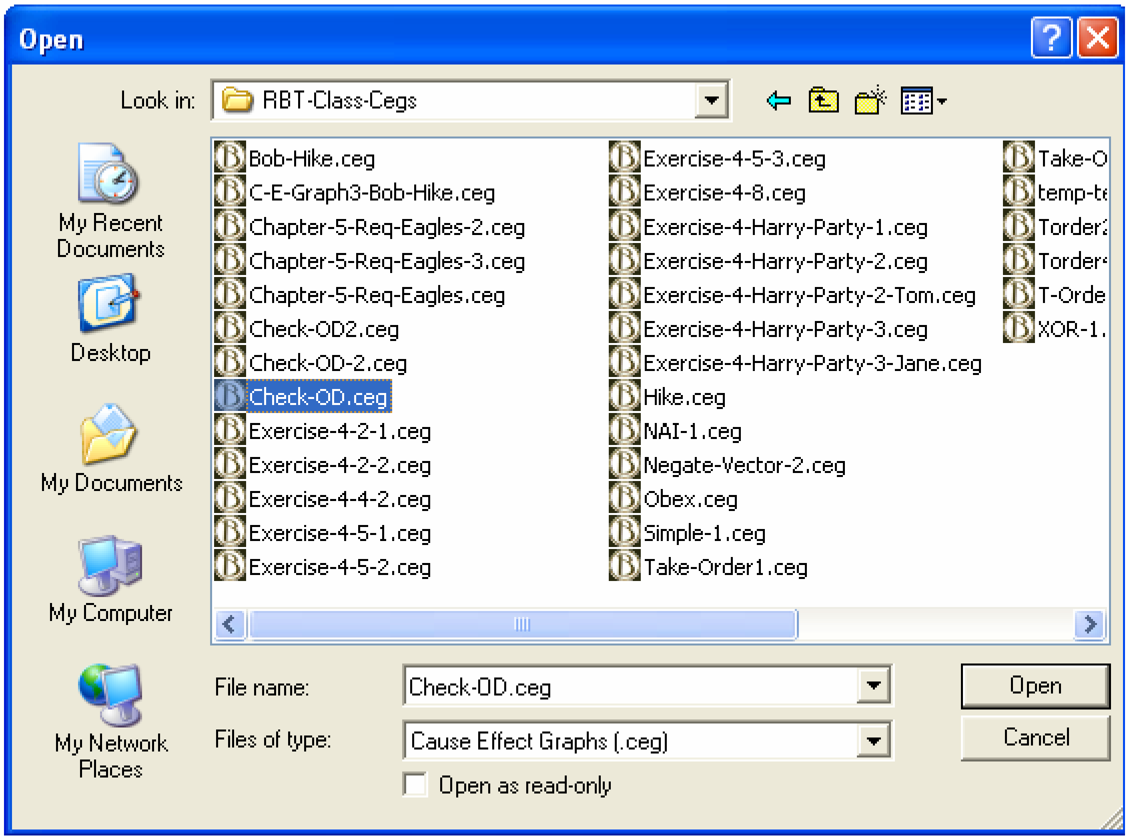


C:\BenderRBT\Documentation\Bender-RBT-Documentation\7.0\RBT-User-Manual\RBT-Graph-Files\Check-OD.rbt

**New Graph Screen**

## 3.2 Importing Existing Graphs From Earlier Versions of RBT

At this point you would also be ready to import any graphs created in earlier versions of RBT. You can do this for graphs created using the Visio front end. You can also do it for graphs that were created using the original character interface, which is now the API referred to above. This is done via the ceg file, not the vsd file. Create a new graph as described in the above section. Then select **File → Import Ceg**. A dialog will appear asking for the ceg file name.



**Import CEG File**

Double click on the ceg file or select it and hit Open. The graph will be imported into the new graph you just set up.

Any graph created using the Visio front end can imported as is. RBT is fully backwards compatible with those versions. For those importing from ceg files created using the text interface, there are some restrictions. RBT will not import comments, subgraphs, or tests. The tests can be brought in separately via the test management facilities which will be covered later. Also, RBT cannot import relation statements with implicit nodes. For example:

A :- (B or C) and (D or E).

In this statement the (B or C) construct and the (D or E) construct generate implicit nodes. These must be made explicit prior to import. (Note that in the Visio version implicit nodes are impossible to create.) You would have to restructure this into:

I1 :- B or C.
I2 :- D or E.
A :- I1 and I2.

You would also have to add I1 and I2 to the node list. These are called Explicit Intermediate nodes.

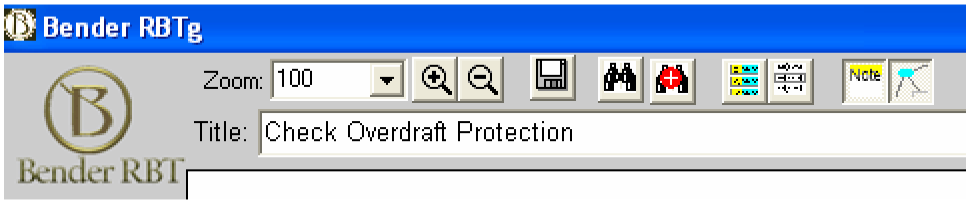
## 3.3 The Title Statement

The first thing to do is to enter a Title for the graph. The information in the Title will be used in the header of all of the reports generated by RBT. This allows you to trace back the generated information (e.g. test case descriptions, test coverage matrix) to the graph from which they were derived.

One guideline to follow is to enter in the name of the function and its version number as the Title. Then if the requirement is updated you will know which version was used to create the graph and the tests.

The Title statement may be up to 1020 characters long and contain any character except the single quote. The constraint is a limitation imposed from the API discussed above.

In this case we will give our graph the Title "Check Overdraft Protection".

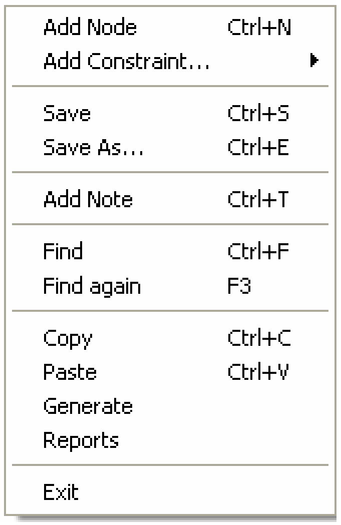


Title: Check Overdraft Protection

**Filling in the Title**

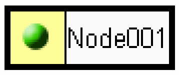
## 3.4 Defining Nodes

You are now ready to start adding in the nodes. To add a node place the cursor in the white working space and right click on the mouse. A dialog will appear giving you a number of choices, including Add Node.



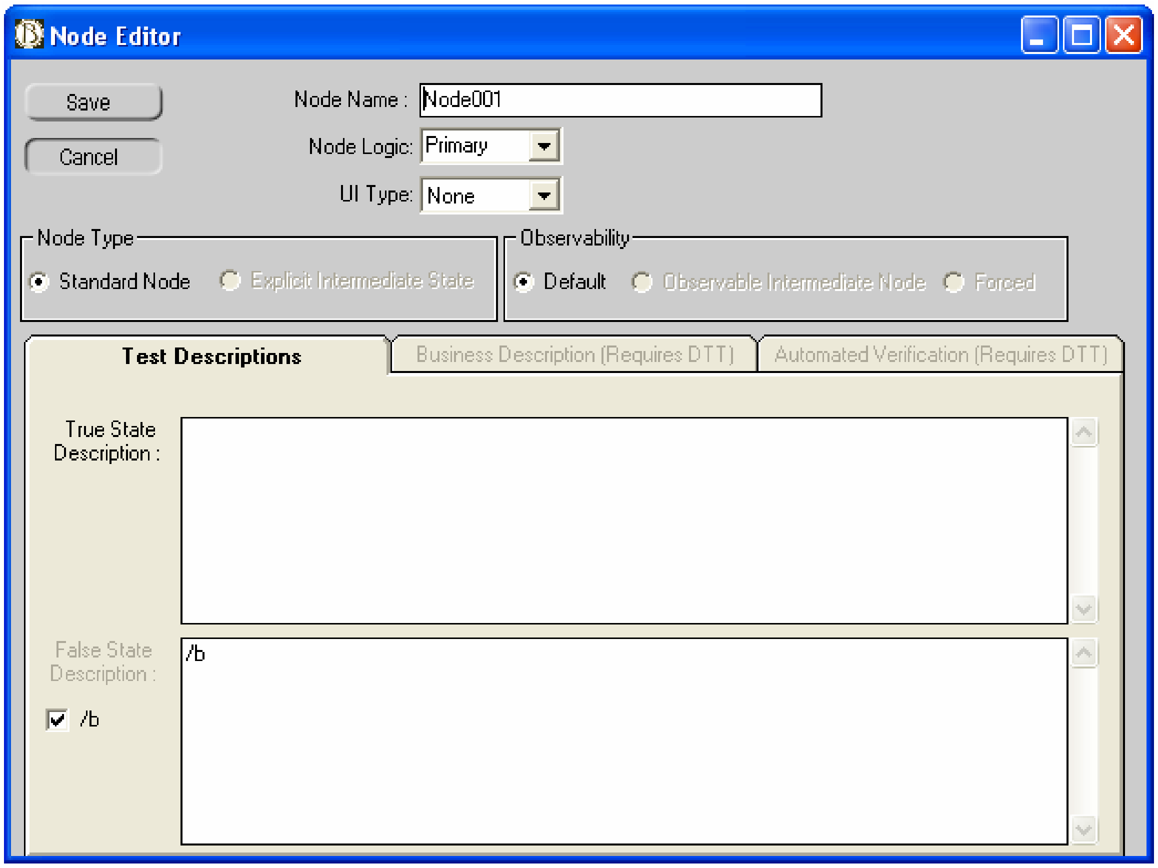
**Add Node**

When you select Add Node a new node will appear on the screen:



**New Node**

Double click on the white portion of the new node and a dialog will appear in which you define its properties:


Node Editor
Save
Cancel
Test Descriptions
Business Description (Requires DTT)
Automated Verification (Requires DTT)

Node Name : Node001
Node Logic: Primary
UI Type: None

Node Type: [✓ Standard Node] Explicit Intermediate State

Observability: [✓ Default] Observable Intermediate Node Forced

True State Description :

False State Description : /b

[✓] /b

**Node Editor**

The Node Editor dialog contains a number of fields for the user to fill in: Node Name, Node Logic, True State Description, False State Description, Node Type and Observability.

[Note: There are additional fields in the dialog – UI Type, Business Description, Automated Verification – that are active only if Direct To Test (DTT) is installed. DTT is an add-on which extends the capability of RBT to include generating the actually executable test scripts. It can generate running scripts in WinRunner, SILK, ROBOT, and other playback tools. DTT falls under the general category of a Framework tool.]

### 3.4.1 Node Name

Node Names may be up to 32 characters long. Node names may consist of any mix of these characters:

> A through Z, a through z, 0 through 9 and the seventeen characters:
> ! @ # $ % ^ - _ ? \ " & + < > { }

The following fifteen characters may not be used:

> ( ) [ ] . , ; : | / ' * ` = ~
> It may also not contain spaces.

The restrictions exist because of the parser in the API.

We suggest that you keep names short but meaningful. Do not name things X, Y, and Z. Name them with something that will make it easier to read and understand the Functional Variations report which uses these names.

### 3.4.2 Node Logic

The node logic defines what type of node it is:

> Primary (a primary cause to the graph)
> Simple (an effect that is the result of a simple relation)
> And
> Or
> Xor
> Nand
> Nor
> Xnor

Just access the pull down and select the option.

### 3.4.3 True State Description

The True State Description defines what you want to display in the test case descriptions when this node is true. These may be up to 1020 characters long. There are no restrictions on the characters included in the description.

Take care to make this as readable as possible. Ideally, you should be able to take these test cases to the user/customer and other domain experts for review (this in effect moves user acceptance test up prior to the start of coding).

One advantage of this is that all of the tests will read exactly the same. This avoids having such things as “it was a valid log-on”, “the log-on passed”, the log-on was OK” all meaning the same thing. You define once what the description should be. RBT then uses that description for all tests where this node is in the true state. If you later decide to change the wording, you only change it one place. RBT will then update all of your tests with the new description.

There are times when you do not want to have anything in this field – e.g. a dummy intermediate node. Just type in a space.

### 3.4.4 False State Description

The False State Description works the same way as the True State Description. The default is the “/b” which means do not display anything when it is false. (You may also uncheck the /b box and enter a space.)

If this node is one of a list of related nodes such that if one of them is true the others will be false, then use the /b option. For example, let us say that we have three customer types: corporate, retail, and government. If one of them is true, the others will be false. We do not want the test case to read: the customer is a corporate customer, the customer is not a retail customer, the customer is not a government customer. We just want it to say: the customer is a corporate customer.

Such tightly coupled nodes should be in an Exclusive, One, or Inclusive constraint. If they are in an Exclusive constraint and you get the test where all of the causes are false, RBT will override the /b and insert a full negative description automatically for all of the nodes.

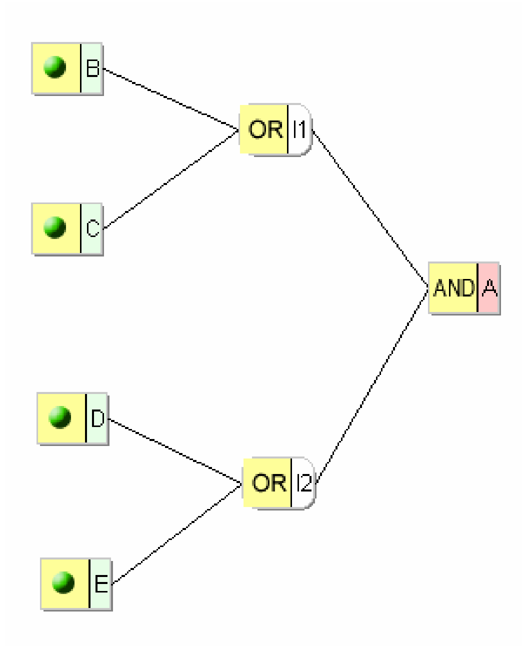
The other case where you would generally leave the False State blank is for an Explicit Intermediate node.

### 3.4.5 Node Type

The choices for Node Type are Standard Node and Explicit Intermediate Node. The Standard Node is the default. It just means that the node represents a cause or an effect that has specific meaning in modeling the requirement in the graph.

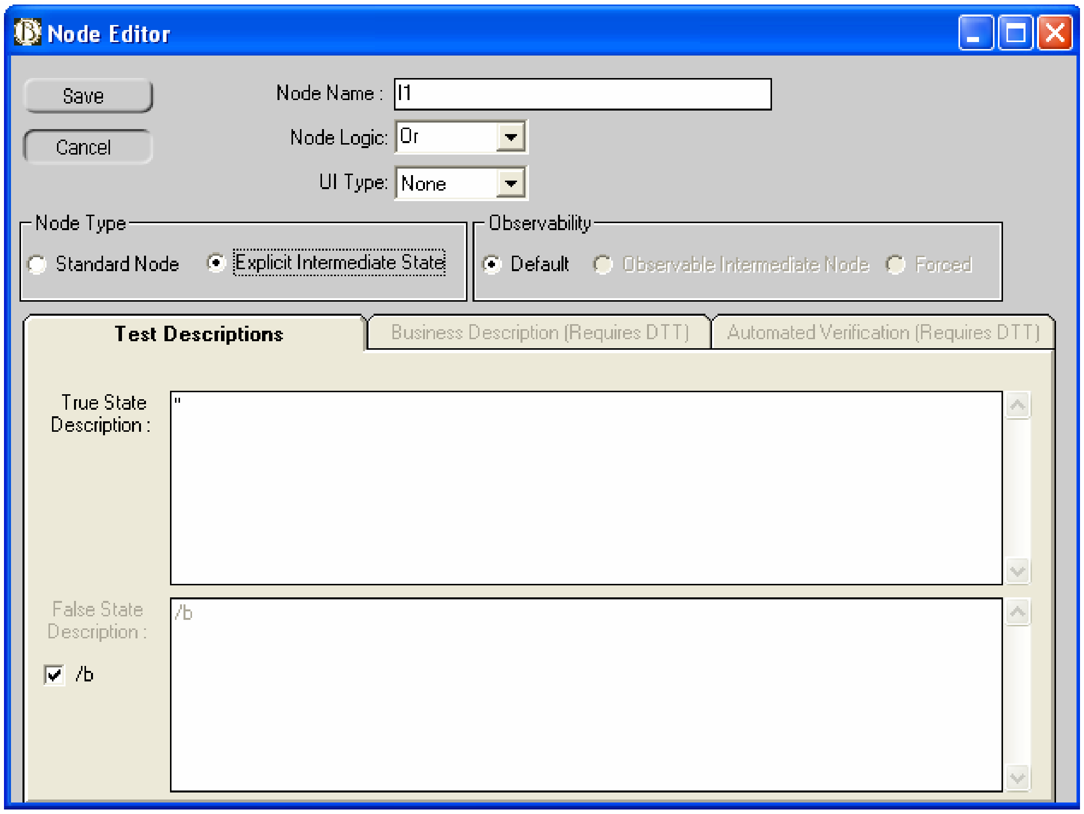
The Explicit Intermediate Node, also called a dummy intermediate node, is the result of graphing a compound logical expression. For example, the specification says that A is true if (B or C) is true and (D or E) is true. To draw this you need a node that represents (B or C), another that represents (D or E), and then use an AND to link them to A.

The graph would be:



**Explicit Intermediate Nodes**

Nodes I1 and I2 are Explicit Intermediate Nodes. I1 would be defined as:


Node Editor
Save
Cancel
Test Descriptions
Business Description (Requires DTT)
Automated Verification (Requires DTT)

Node Name : I1

Node Logic: Or

UI Type: None

Node Type: ( ) Standard Node [✓ Explicit Intermediate State]

Observability: [✓ Default] ( ) Observable Intermediate Node ( ) Forced

True State Description : "

False State Description : /b

☑ /b

**I1 Node Definition**

### 3.4.6 Observability

Observability means that someone running a test would be able to directly see the state of the node. Things that are inherently observable are objects on a screen, updates to a data base, packets sent over the communications lines, and objects on reports. Some systems also have additional observable objects such as sound and movement (e.g. robotic arms).

RBT assumes that primary effects are observable and that intermediate nodes are not observable. This has a significant impact on designing tests. We need to ensure that we not only get the right answer but that we get the right answer for the right reason. If an intermediate node is not observable and a defect occurs at that point, the defect must be propagated to an observable point. This propagation must be done in such a way so that two or more defects cannot cancel each out. It must also be done in such a way so that something going right on one part of the path does not hide a defect from another part of the path. This is called sensitizing the test path. It is taken care of by the test design engine.

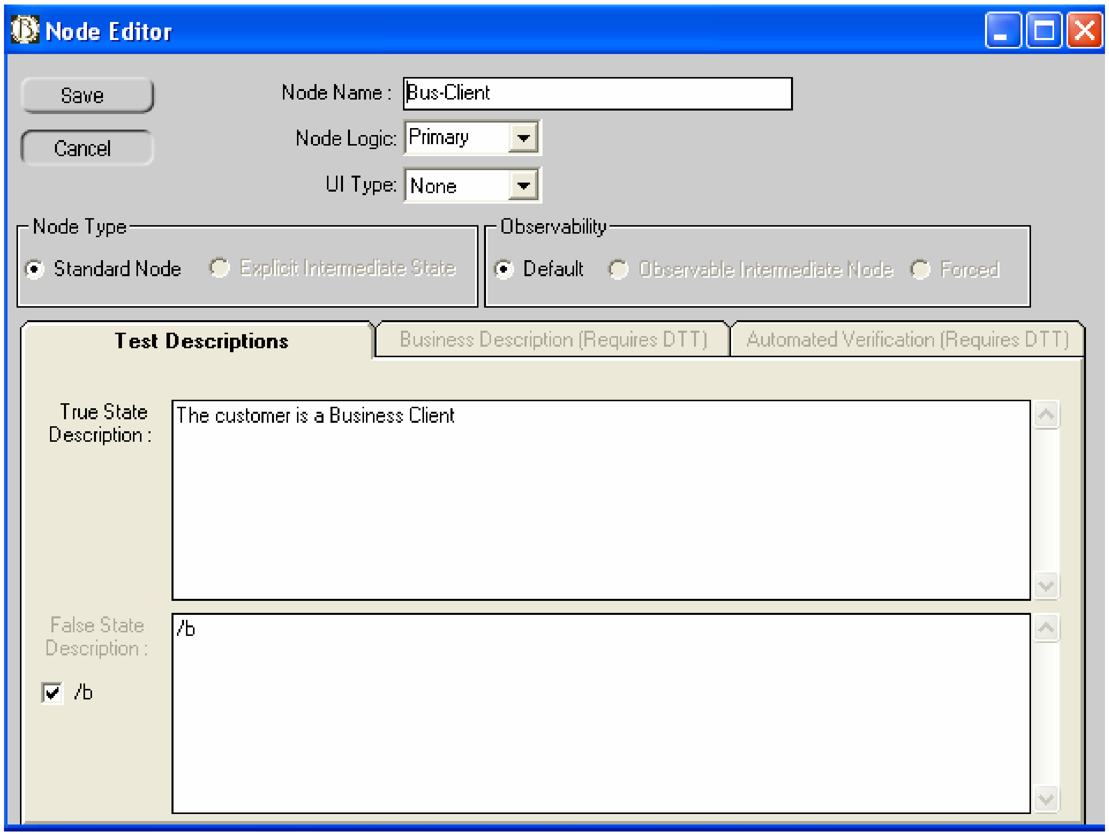
If an intermediate node is not observable, then leave the setting to the default.

If an intermediate node is actually observable, then set the node to an Observable Intermediate Node.

When RBT is designing the tests it is sometimes impossible to sensitize the path. This can occur because of constraints and/or the overall logic. The result will be that some of the Functional Variations will be flagged as Untestable. This means that they are legitimate variations that must be tested but that there is no way to sensitize the path given what is normally observable. To solve this you must force the intermediate node to be observable – i.e. create a diagnostic probe point in the code. You note this by setting the Forced choice. The test cases will be annotated to remind you that the test relies on a diagnostic being created.

## 3.4.7 Check OD Example – Node Definitions

Now let us get back to our Check Overdraft problem. The first node to create is the business client node.


Node Editor
Save
Cancel
Test Descriptions
Business Description (Requires DTT)
Automated Verification (Requires DTT)

Node Name : Bus-Client

Node Logic: Primary

UI Type: None

Node Type: [✓ Standard Node] Explicit Intermediate State

Observability: [✓ Default] Observable Intermediate Node Forced

True State Description : The customer is a Business Client

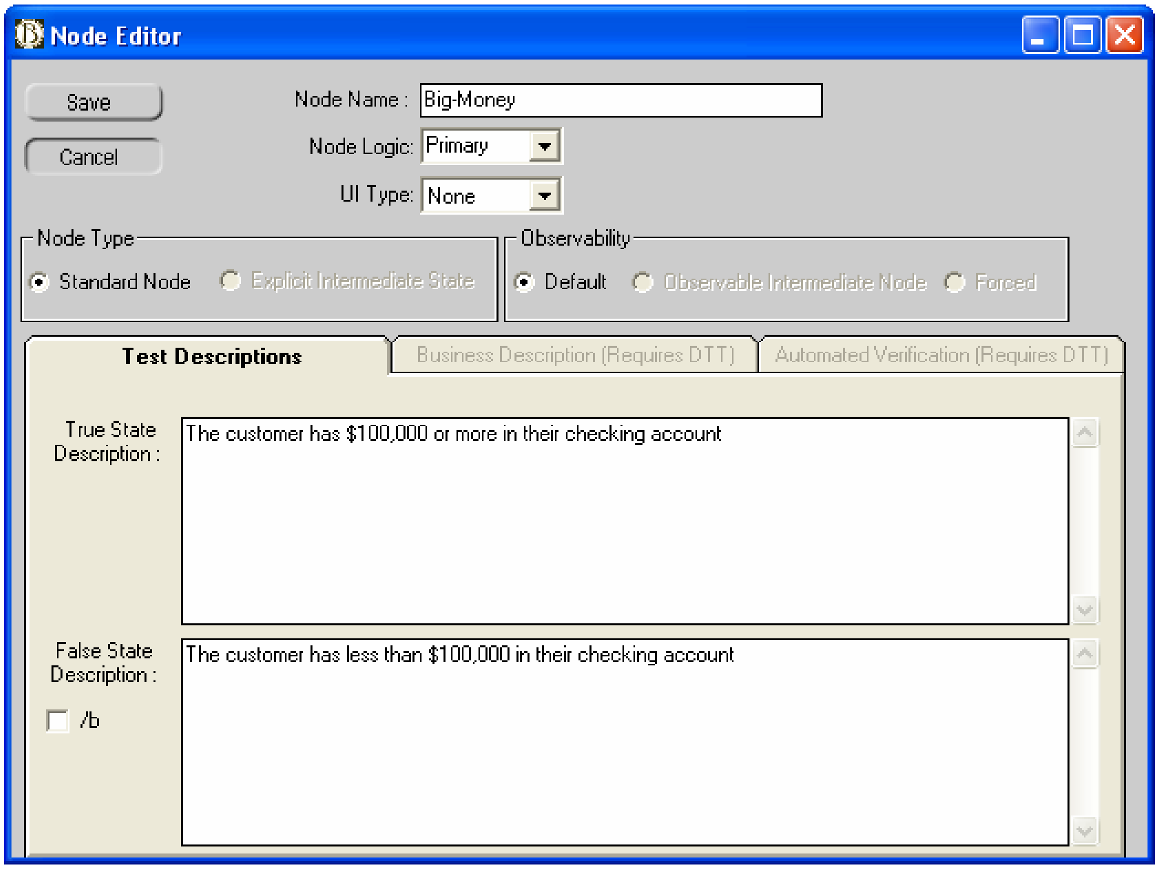
False State Description : /b

[✓] /b

**Business Client Node**

Notice that the false state is defined as blank. This is because we have another, mutually exclusive customer type.

Another node to define relates to the amount of money in the account.


Node Editor
Save
Cancel
Node Type
Explicit Intermediate State
Observability
Observable Intermediate Node
Forced
Test Descriptions
Business Description (Requires DTT)
Automated Verification (Requires DTT)

Node Name : Big-Money

Node Logic: Primary

UI Type: None

Node Type: [✓ Standard Node]

Observability: [✓ Default]

True State Description : The customer has $100,000 or more in their checking account

False State Description : The customer has less than $100,000 in their checking account

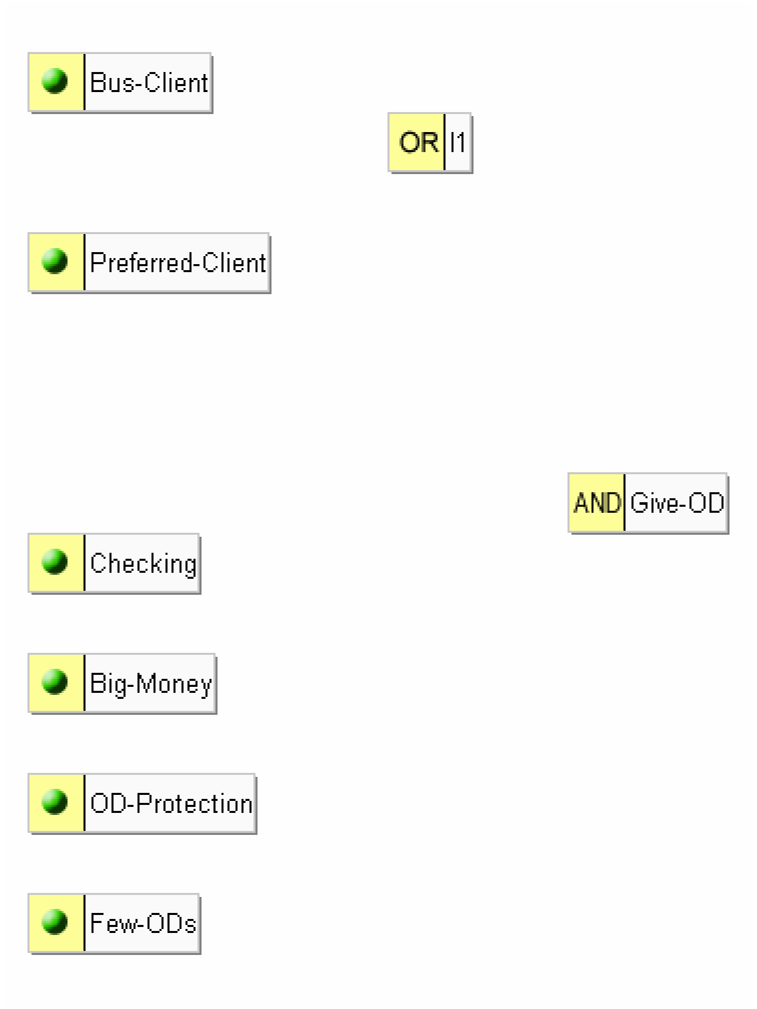
☐ /b

**Account Balance Node**

In this node there is a description for the false state.

## 3.5 Linking The Nodes Into A Graph

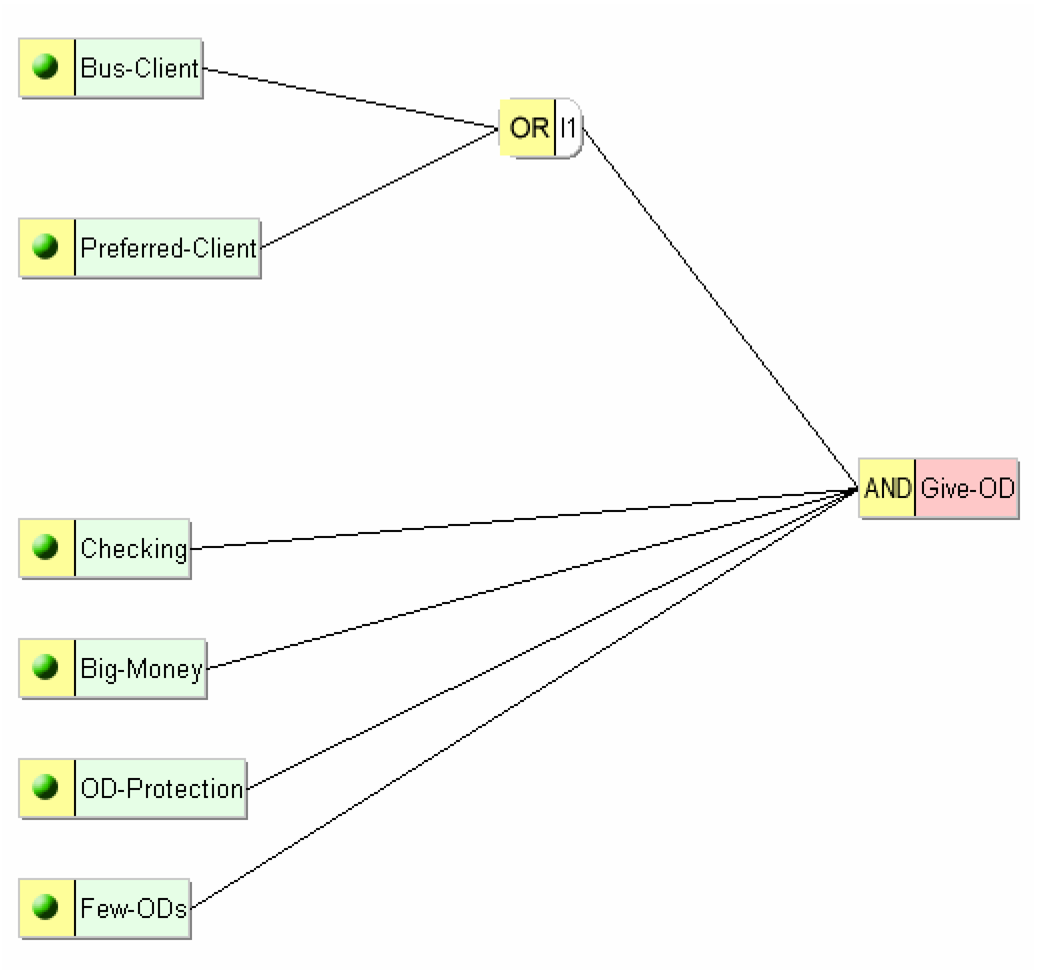
When we have defined all of the nodes to RBT we would be at this state:



**Check OD Nodes**

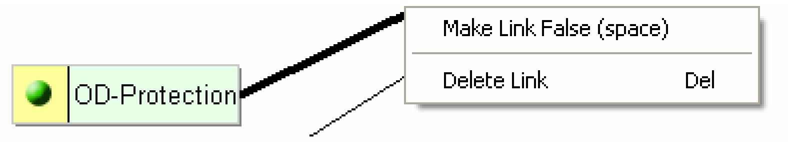
We need to link them together. Linking is always from → to; that is you link from the cause to the effect. Graphs are meant to be read left to right. For example, you need to create a link from the node "Checking" to the node Give-OD. To do this you double click on the left hand portion of the node (the yellow box). When you do this the word "linking" will appear. You then click on the target node and a line is created linking them. The lines connecting the nodes are called vectors.

After we created the links the graph would look like this:



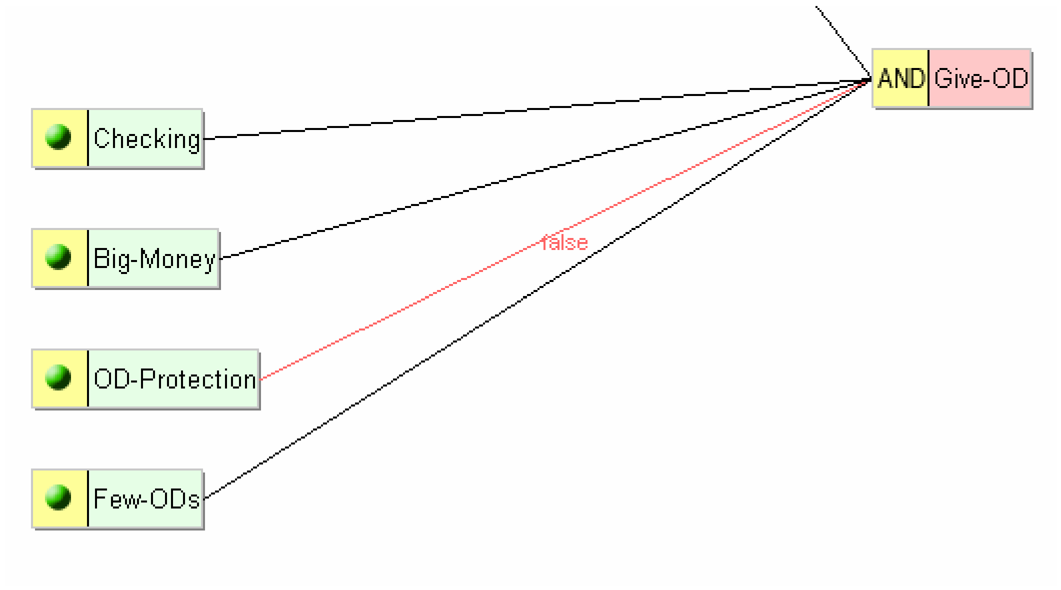
**Check OD Graph**

We still need to adjust something in the graph. Part of the rule is that you must NOT have overdraft protection already in order to get the free overdraft protection. We need to change the vector from true (the default) to false. To do this, select the vector by putting the mouse on it and clicking. Then do a right click. A dialog will pop up which allows you to toggle the state of the vector between true and false.



**Toggle the Vector to False**

Once you have done that the vector will change color to red and the word "false" will be add to the line. That portion of the graph now looks like:

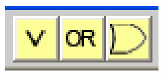


**False Vector**

The nodes of the graph are now all linked together. The next step is to add the constraints.

### 3.5.1 Selecting the Logic Symbols

RBT gives you three choices for the logic symbols in the graph: classic logic symbol, words, and electrical engineering symbols. To choose the type you want just select it from the menu bar:

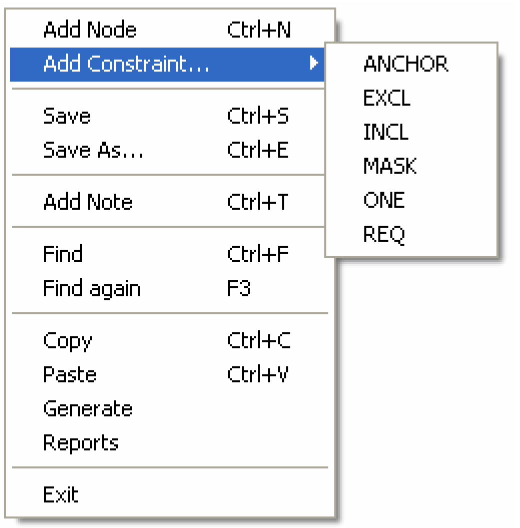


**Cause-Effect Graphing Logic Symbols**

When you select your desired symbol set the graph will be immediately updated.

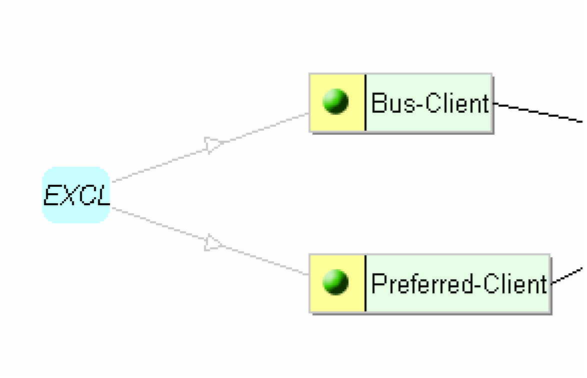
## *3.6 Adding the Constraints*

To add constraints to the graph position the mouse to the work area and right click (just as you did in adding a node). Select Add Constraint and select the type of constraint you want.



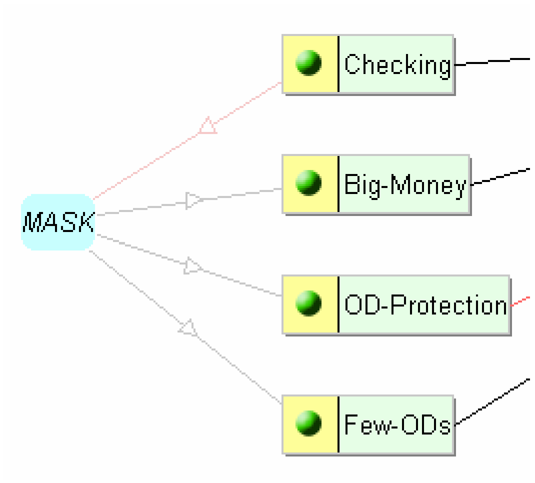
**Add Constraint Menu**

The selected constraint will appear on the graph. You then need to connect it to the desired nodes. Do this by double clicking on the constraint. The word "Linking" will appear. Then click on the nodes in the constraint. If the constraint is a bi-directional constraint (Exclusive, Inclusive, One) then the order of selecting the nodes does not matter.



**Bi-Directional Constraint**

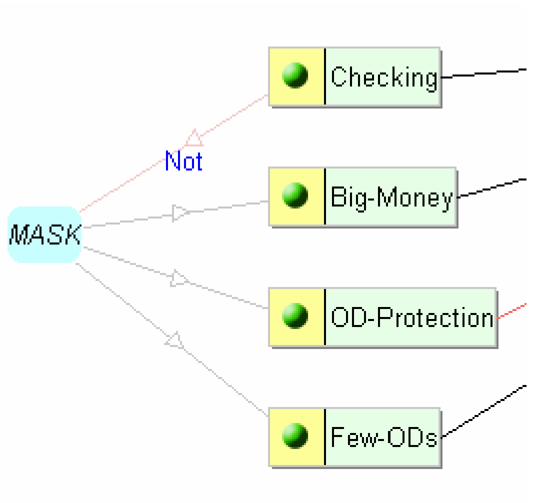
If the constraint is a unidirectional constraint then the SUBJECT of the constraint must be selected first and then the OBJECTS. In our Check-OD example we have an attribute Mask with Checking as the Subject and the Big-Money, OD-Protection, and Few-ODs as the Objects. In other words, if you do not have a checking account how could you have a lot of money in it or no overdraft protection for it? In adding this constraint we first select the Mask, double click on it to activate linking, and then select Checking first. After Checking has been selected, select the other Objects.



**Uni-Directional Constraint**

On the graph notice the direction of the arrows. The arrow flows from the Subject node and to the Object nodes. Also the arrow from the Subject node is red, while those to the Objects are blue.

We still have one more step in setting up the Mask. In this case it is when we do NOT have a Checking Account that the other nodes are masked. Just as in toggling a vector in a relation statement we can also toggle the vector in a constraint. Just select the vector, right click, and select the make false option. Our Mask constraint now looks like this:

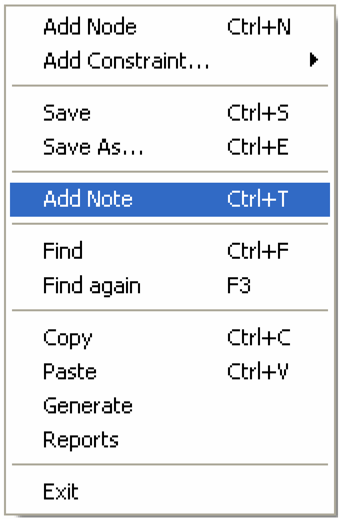


**Uni-Directional Constraint – False Subject**

For any constraint you can make one or more of the connections False. For example, the constraint might be that A, B, and NOT C are mutually exclusive. You would just add an EXCL constraint to the graph, link nodes A, B, and C to it, and make the vector connecting the EXCL to C false.

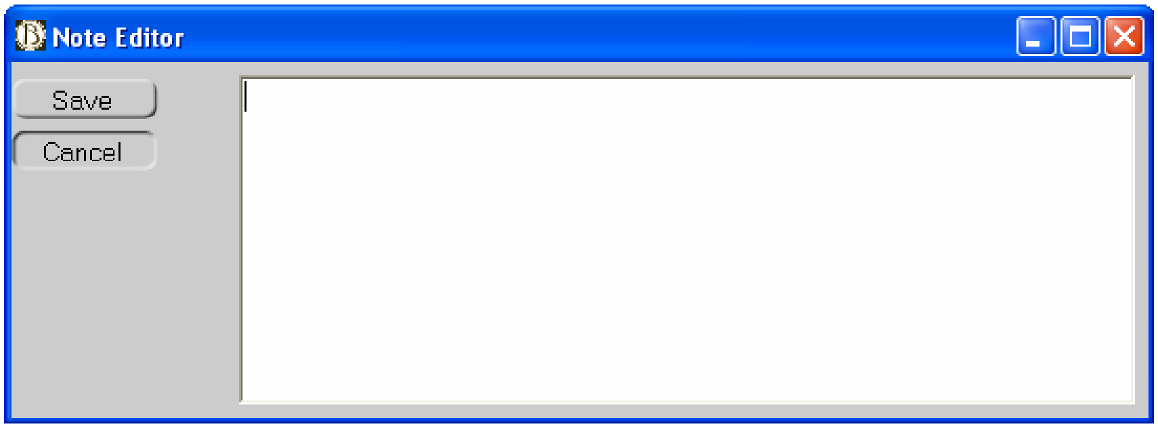
## 3.7 *Adding a Note*

You can add notes to the graph – i.e. comments. To do this place the mouse in the working area, right click, and select the Add Note option:



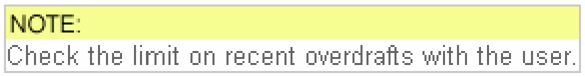
**Add Note**

When this is selected the Note Editor will appear. This is a free form comment block. There are no restrictions on the characters. The comment may be up to 8190 characters long.



**Note Editor**

Type in your note and hit Save. The note will appear on the screen.

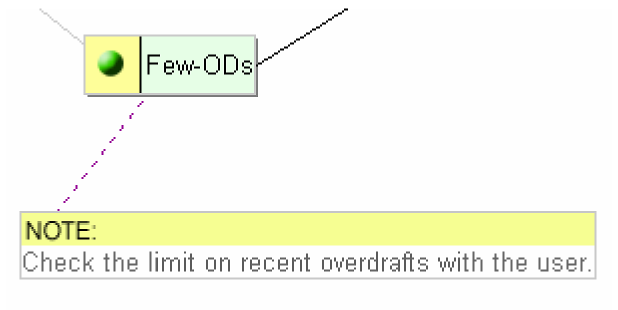


NOTE:
Check the limit on recent overdrafts with the user.

**Note Example**

After the note has been created, you can edit it later. To do this, double click on the white portion of the Note. The Note Editor dialog will appear again. Make your changes and select Save.

Notes may be free floating. They may also be connected to one or more nodes and/or constraints. To connect a note to a node and/or constraint, double click on the yellow portion of the Note. The "Linking" message will appear. Click on the target node/constraint and a dotted line connecting the note to the object will appear.



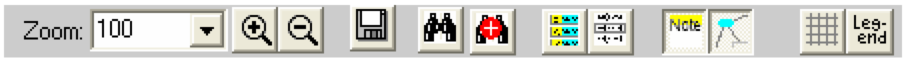
NOTE:
Check the limit on recent overdrafts with the user.

**Note Example – Connect to Node**

You can later disconnect the note by selecting the dotted line and hitting Delete.

## *3.8 Miscellaneous RBTg Utilities*

There are a number of useful utility functions to aid in drawing the graphs. These are accessed by the buttons across the top. We will cover them in the order they appear from left to right.

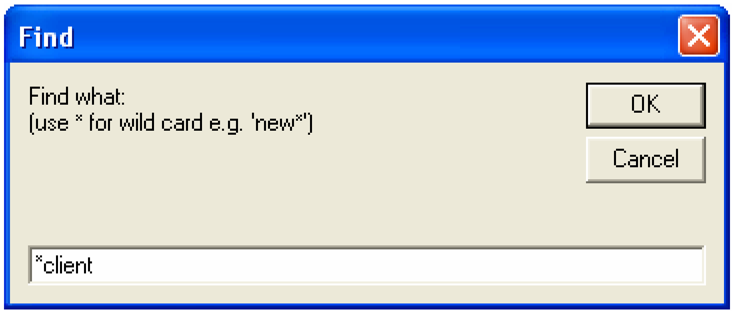


**RBTg Utilities**

**3.8.1 Zoom** – The zoom pull down and the magnifying glasses with the + and – symbols allow you to make the drawing larger or smaller on the screen.

**3.8.2 File** – The File symbol is a Save function.

**3.8.3 Find / Find Again** – The binoculars allow you to find a specific node in the graph. This can be very useful in large graphs with over a hundred nodes. Selecting this option results in the Find editor appearing:



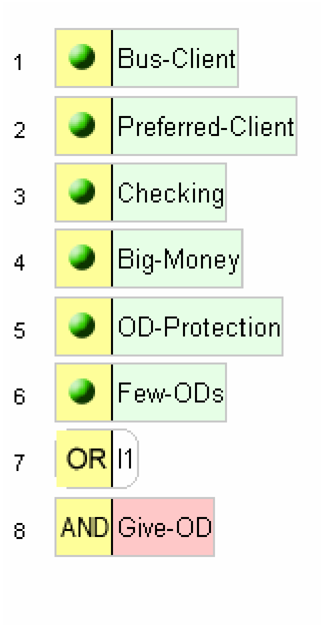
*client

**Find Editor**

Type in the node you are looking for. You may also use wild cards. This is done via using an *. When you hit OK it will take you to that node. If a wild card was used and more than one node meets the search criteria it will take you to the first instance of a match.

If you select the binoculars with the + sign (Find Again) it will take you to the next instance on the graph. In this example, by entering in “*client” it would first take us to Bus-Client. Hit the Find Again button would then take us to Preferred-Client.

**3.8.4 Node List** – Selecting the Node List option will display the list of nodes in your graph in list form in a window on the left side of the graph:



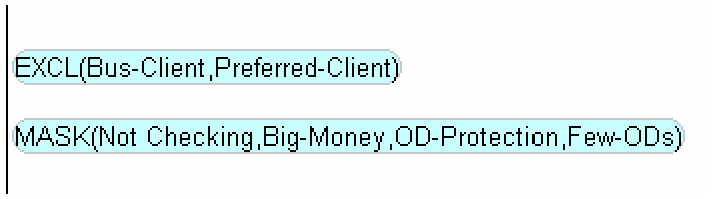
**Node List**

If you select a node from this list it will also highlight that same node on the graph. This is useful in finding a node quickly. Double clicking on the node in the list will bring up the Node Editor so that you can modify its definition.

Another use of this list is to tune the order in which the causes and effects appear in the test scripts. You can grab the node and move it up or down in the list. This will change the order in the tests generated.

### 3.8.5 Constraint List

– Similarly you can display in list form the set of constraints. Selecting one will highlight the corresponding constraint in the graph.


EXCL(Bus-Client,Preferred-Client)
MASK(Not Checking,Big-Money,OD-Protection,Few-ODs)

**Constraint List**

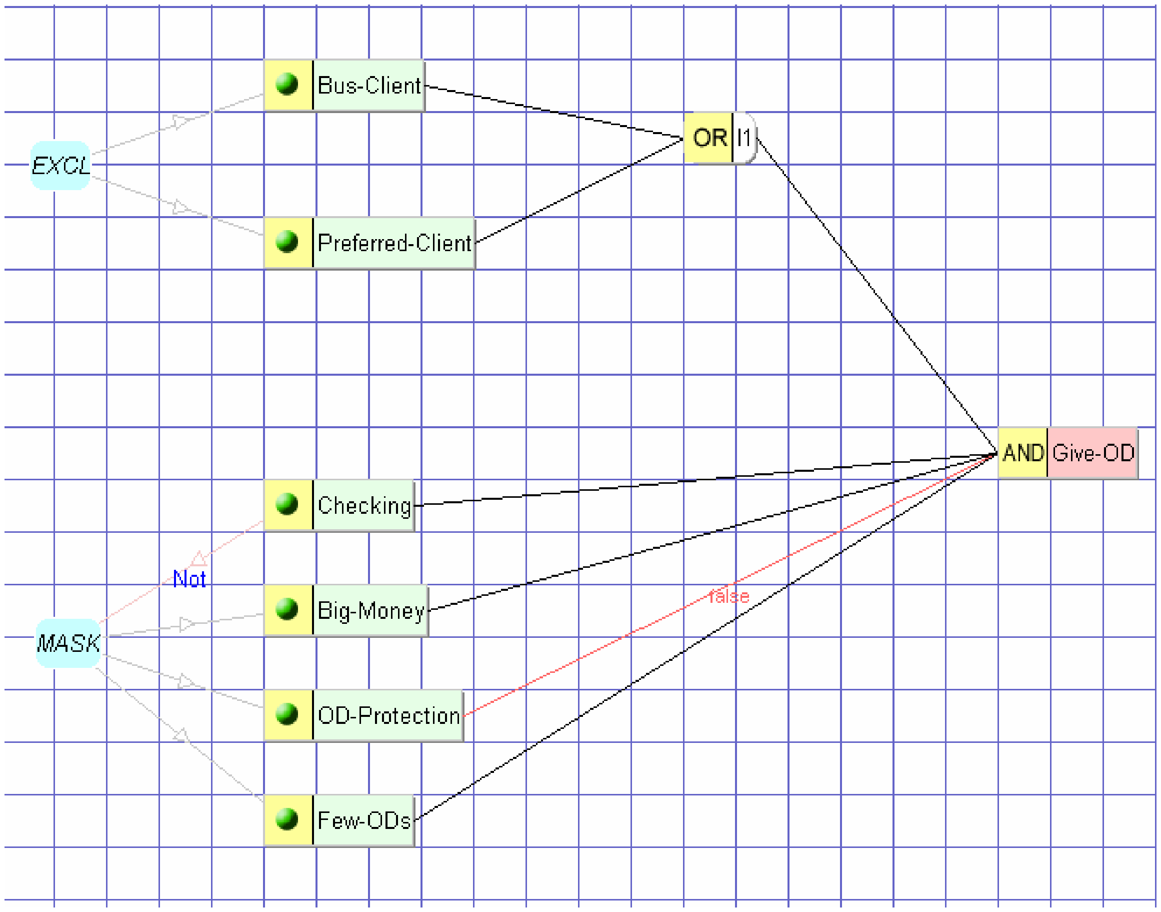
### 3.8.6 Note Display

– The Note Display option allows you to toggle on and off the display of any notes for the graph.

### 3.8.7 Constraint Display

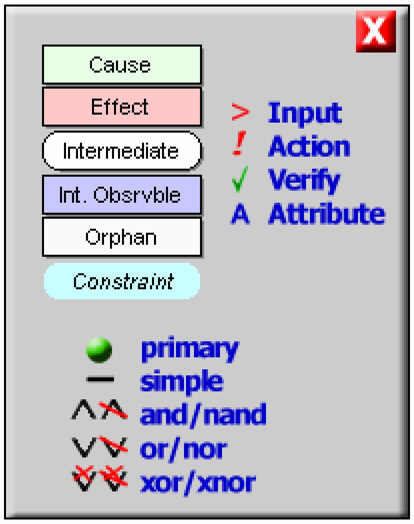
– The Constraint Display option allows you to toggle on and off the display of the constraints on the graph. This is very useful when you have a complex graph with many constraints, especially large numbers of Masks which involve many of the same Nodes. If you have the display of the constraints set to off you can add another constraint and only that one will be displayed. Each additional constraint added will also be displayed until you cycle through the display / don't display constraint option.

**3.8.8 Display Grid** – The Display Grid option adds a grid to the work area and also automatically lines up the nodes on the graph to the closest grid cell.



**Graph With Grid On**

**3.8.9 Legend** – The Legend option displays the color coded legend for the objects in the graph work area:



**Legend**

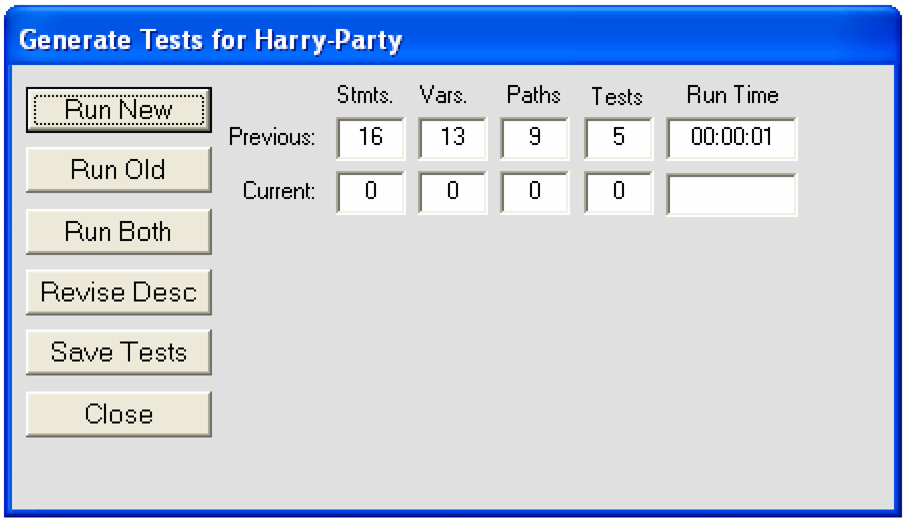
The shape and color of an object denotes what kind of object it is – e.g. a primary cause is displayed as a light green rectangle, an orphaned node (i.e. not yet part of any relation) is displayed as a white rectangle. The Input, Action, Verify, and Attribute symbols apply only to the DTT version of RBT.

# 4. Creating and Managing Test Cases

The Cause-Effect Graphing process is an iterative one. You generally graph, review the results, and tune the graph until you are sure the requirements are solid and that the graph reflects those requirements. You then implement the test cases. When you commit to building the executable tests you want to ensure that RBT knows that this set of tests is the one you are implementing. This will allow you to protect your investment in these tests.

If RBT if aware of existing tests, it can evaluate those tests as the requirements and graph change. How much coverage do the old tests give you? What new tests will you need? What modifications have to be made to the old tests? RBT can answer those questions for you.

Therefore, RBT gives you a number of options in generating test cases. From the main menu select Generate. You will get the following screen:



| | Stmts. | Vars. | Paths | Tests | Run Time |
|---|---|---|---|---|---|
| Previous: | 16 | 13 | 9 | 5 | 00:00:01 |
| Current: | 0 | 0 | 0 | 0 | |

**Test Generation Options**

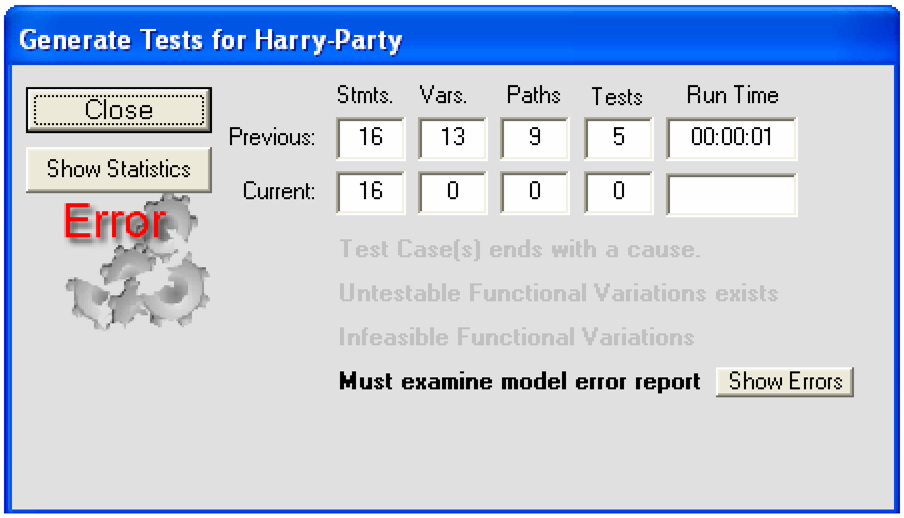
The **Run New** option will design a new set of tests based on the graph you have just entered.

The **Run Old** option will evaluate the coverage of a set of existing tests against the current version of the graph.

The **Run Both** option will evaluate the coverage of a set of existing tests and then supplement these tests to complete the coverage of the graph.

The **Revise Desc**[riptions] option allows you to modify the True or False definition of a node without having to rerun the graph. It just updates the data base with the new description. This is immediately reflected on all of the generated reports (e.g. Batch Test Cases). This is very convenient in cases where you have a long running graph and find that you want to tune the wording on a description. For example, you may have misspelled something or you want to bring the wording to be more in sync with other sets of tests.

You may also use this feature when you change the wording in the Title statement. However, if you add, delete, or modify any constraints and/or relation statements, this function will reject with an error.

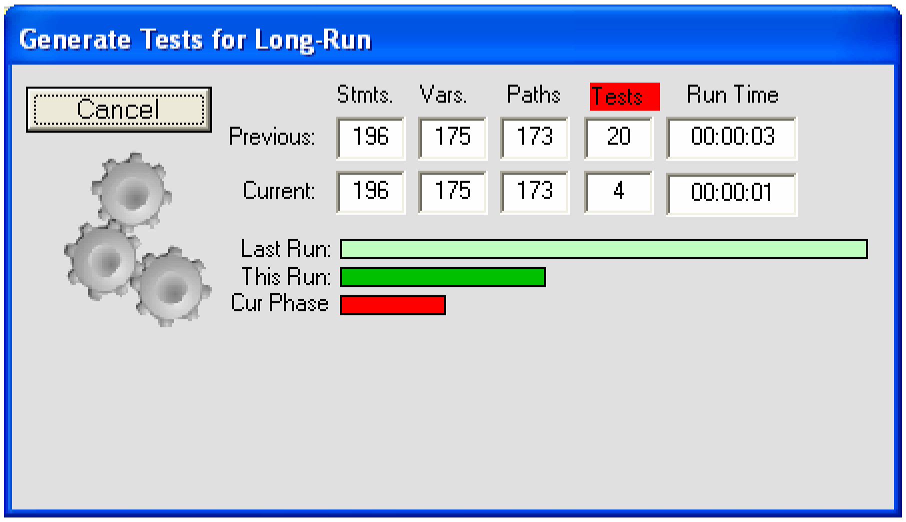


|  | Stmts. | Vars. | Paths | Tests | Run Time |
|---|---|---|---|---|---|
| Previous: | 16 | 13 | 9 | 5 | 00:00:01 |
| Current: | 16 | 0 | 0 | 0 |  |

Test Case(s) ends with a cause.
Untestable Functional Variations exists
Infeasible Functional Variations
**Must examine model error report**

**Change Descriptions Error**

Most test design runs will finish in under a second or so. However, if the run last longer than that you will see a progress thermometer appear:



|  | Stmts. | Vars. | Paths | Tests | Run Time |
|---|---|---|---|---|---|
| Previous: | 196 | 175 | 173 | 20 | 00:00:03 |
| Current: | 196 | 175 | 173 | 4 | 00:00:01 |

**Run Progress Thermometer**

**“Statements”** is the number of statements in the generated API syntax.
**“Vars”** is the number of functional variations identified.
**“Paths**” is a construct internal to the test design engine.
**“Tests”** is the number of test cases designed to cover the variations.
**“Run Time”** for the previous run is the total time it took to completion. For the current run it is the elapsed time so far.

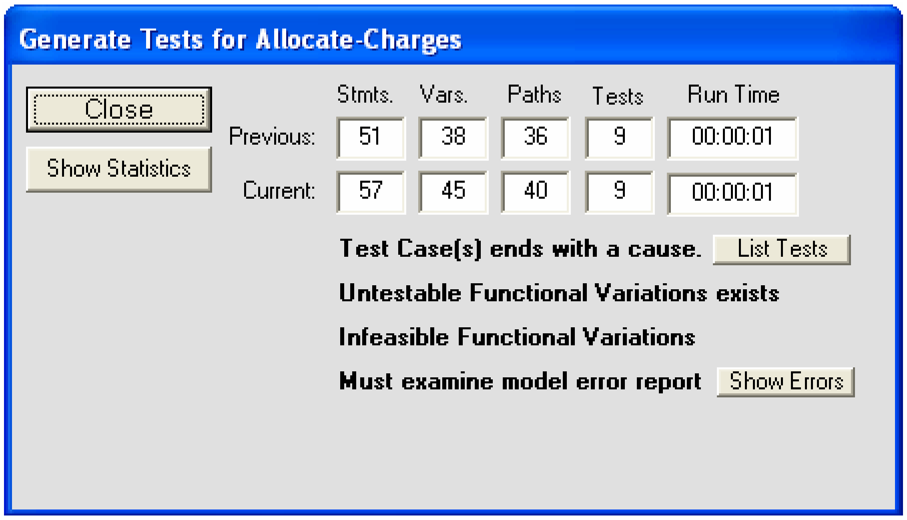
## 4.1 *Creating a New Set of Test Cases*

Once you have drawn the graph you are ready to have RBT design an optimal set of test cases which completely cover all of its functionality. From the RBTg graphing interface select:

Generate → Run New

If there are hard errors in the graph you will see the error message just as in the Change Descriptions Error example. Review the Error Report to determine what you need to correct.

If it runs correctly you will see an update to the generate tests dialog:


Generate Tests for Allocate-Charges

|  | Stmts. | Vars. | Paths | Tests | Run Time |
|---|---|---|---|---|---|
| Previous: | 51 | 38 | 36 | 9 | 00:00:01 |
| Current: | 57 | 45 | 40 | 9 | 00:00:01 |

**Test Case(s) ends with a cause.**
**Untestable Functional Variations exists**
**Infeasible Functional Variations**
**Must examine model error report**

**Generate Tests Dialog – Completed Run**

One or more of the following messages may appear:

**“Test Case(s) ends with a cause.”** This means that in the Script version of the test cases (see Reports Section) that one or more of the test cases ends with only the cause portion of the test described. This is because the last effect was defined with a null false description. If the tests are really Batch tests then you can ignore this. If this is a test of an interactive dialog then you need to refine your node definitions to fix this.

**"Untestable Functional Variations exists."** RBT was not able to sensitize the test path for one or more variations in such a way as to ensure that any possible defect would show up at an observable point. Such variations are valid variations which must be tested but are not included in any test yet. To solve this, first ensure that all naturally observable intermediate nodes have been flagged as such in the node descriptions. If this does not eliminate all of the untestables you must then identify points to force observability – i.e. you need to insert a diagnostic probe point in the code to ensure that you get the right answer for the right reason. Set the appropriate intermediate nodes to Force Observable.

**"Infeasible Functional Variations."** One or more functional variations have been flagged as infeasible because they violate constraint(s) and/or the overall graph logic. These will be identified on the Functional Variations Report (see Reports Section). Review each one to ensure that it is legitimately infeasible. If the variation should be feasible there is either a problem with the graph or a logic error in the Requirements Specification which the graph was derived from.

**"Must examine model error report."** Running the graph resulted in one or more items on the Errors Report (see Reports Section). If the graph ran to completion, these are mostly warning messages. You can see the report by hitting the Show Errors button or via the Reports button on the main menu.

## *4.2 Saving Your Tests*

After you have reviewed your tests and are ready to start implementing them you want RBT to remember this set of tests. The objective is to protect your investment in these tests. It generally takes three to five times the effort to build an executable test than it does to design the test. By having RBT remember this set of tests, you do not have to start all over again when you change the requirements and update the graph.

The first step in this process is to save the tests you have designed and are preparing to implement. You can do this in one of two ways. From the RBTg graphing interface select:

Generate → Save Tests

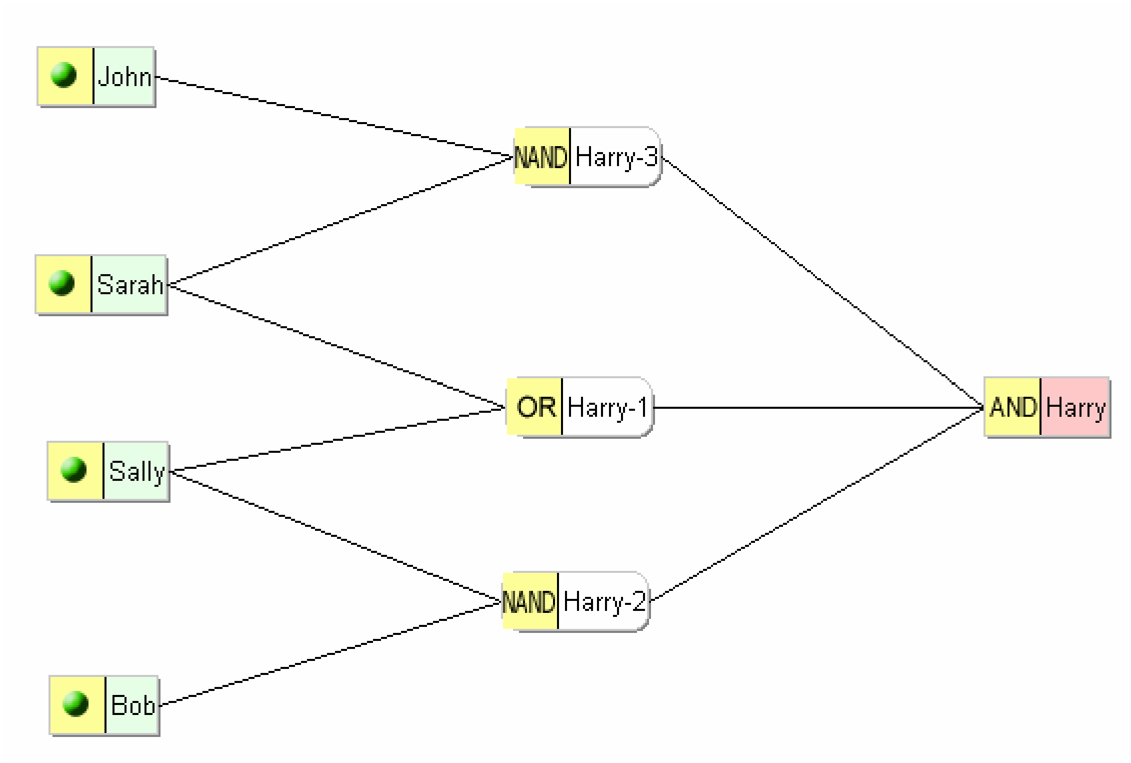
This will bring up a standard save dialog asking you what you want to call the file containing the test definitions and where you want to save it. RBT saves it as a .CET file.

The other path to this feature is from RBT itself. To do this select:

Utilities → Preserve Tests → Save Tests As

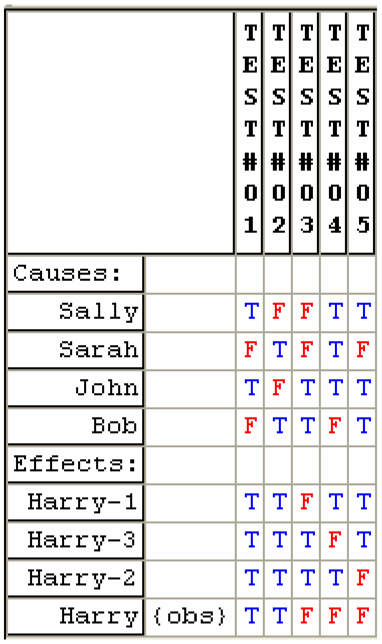
Let's take an example. We will use the Harry-Party.rbt graph from the Examples Directory. The specification for this is:

If [either Sally or Sarah go to the party]
And
[Sarah and John do not go to the party together]
And
[Sally and Bob do not go to the party together]
Then Harry will go to the party.



**Harry Graph – Version 1**

The test case matrix for this graph is:



| | | TEST#01 | TEST#02 | TEST#03 | TEST#04 | TEST#05 |
|---|---|---|---|---|---|---|
| Causes: | | | | | | |
| Sally | | T | F | F | T | T |
| Sarah | | F | T | F | T | F |
| John | | T | F | T | T | T |
| Bob | | F | T | T | F | T |
| Effects: | | | | | | |
| Harry-1 | | T | T | F | T | T |
| Harry-3 | | T | T | T | F | T |
| Harry-2 | | T | T | T | T | F |
| Harry | (obs) | T | T | F | F | F |

**Harry Tests – Version 1**

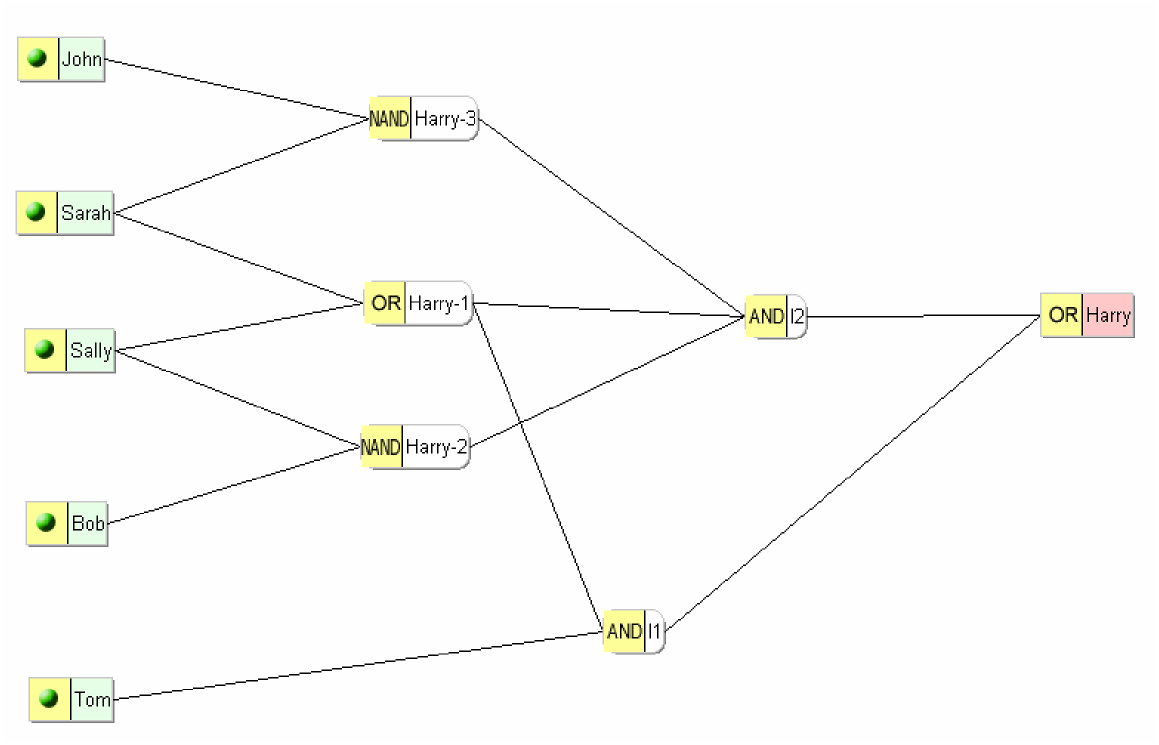
When we save off the tests they have the following format:

TEST#1 = Sally, not Sarah, John, not Bob.
TEST#2 = not Sally, Sarah, not John, Bob.
TEST#3 = not Sally, not Sarah, John, Bob.
TEST#4 = Sally, Sarah, John, not Bob.
TEST#5 = Sally, not Sarah, John, Bob.

There is a Test Case ID which is composed of a generic name and a number. Then the state of only the primary causes is defined. RBT will deduce the states of all of the other nodes from this information and the graph. Each primary cause's name is prefaced with either nothing (i.e. it is True) or "not" (i.e. it is False). If a primary cause is masked for a given test is not mentioned in the set.

## 4.3 Evaluating Old Test Cases

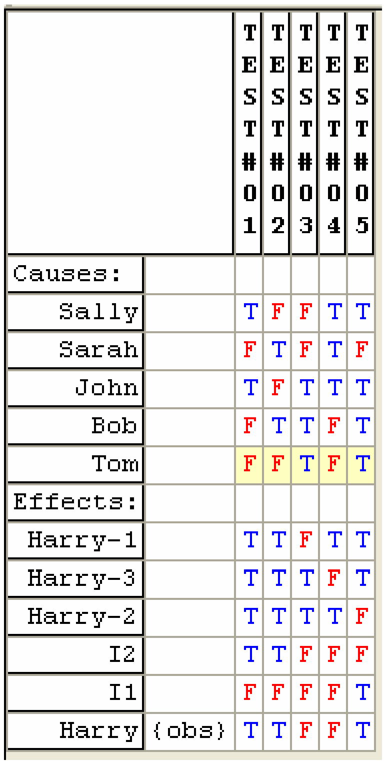
Let us now modify the rules to add in a new variable "Tom". If Tom goes to the party and [either Sally or Sarah go to the party] then Harry will go even if Bob and/or John are there. The new graph is (Harry-Party-2-Tom in the Examples Directory):



**Harry Graph – Version 2**

From the first version we have a set of tests. None of them include Tom. However, we can still use them. RBT will tell us what they cover and what we need to do to bring them in sync with the new application rules, reflected by the graph.

From the RBTg interface choose Generate → Run Old. You will be prompted for the name of the tests file to use. Select the appropriate tests file and run the graph. When we do this for the above modified graph we get the following test definition matrix:



| | | TEST#01 | TEST#02 | TEST#03 | TEST#04 | TEST#05 |
|---|---|---|---|---|---|---|
| Causes: | | | | | | |
| Sally | | T | F | F | T | T |
| Sarah | | F | T | F | T | F |
| John | | T | F | T | T | T |
| Bob | | F | T | T | F | T |
| Tom | | F | F | T | F | T |
| Effects: | | | | | | |
| Harry-1 | | T | T | F | T | T |
| Harry-3 | | T | T | T | F | T |
| Harry-2 | | T | T | T | T | F |
| I2 | | T | T | F | F | F |
| I1 | | F | F | F | F | T |
| Harry | (obs) | T | T | F | F | T |

**Harry Tests – Run Old**

Notice that RBT has added Tom True to some of tests (#3, #5) and added Tom false to some of the tests (#1, #2, #4). RBT uses the old tests as a base and then adds in the new nodes in the optimal manner to maximize coverage under the new rules.

In the Batch and Script Tests Reports this same information is also reflected in the test descriptions. For example, Test #01 would be documented as:

Old Test: TEST#01 -- Harry Goes To The Party - Tom Helps Out

Cause states:
Sally goes to the party
Sarah does not go to the party
John goes to the party
Bob does not go to the party
*Tom is not going to the party

Effect states:
Harry goes to the party

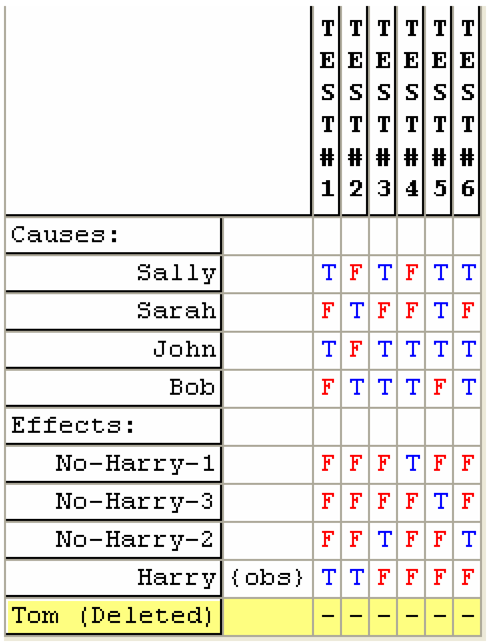
Notice that test is annotated as an Old Test. Also note that the cause state that describes Tom is preceded by an "*" to denote that this is a modification to the old test.

You also need to review the coverage of the Old Tests. In this example, the new graph results in 19 variations. Two of them are marked as Untested – e.g.:

> <NOT-TESTED> T05--Not tested via Old Test Case Definitions
> 9. If Sally and Bob
> then not Harry-2.

"Untested" is different than "Untestable". Untestable means that the variation cannot be included in a test because RBT cannot sensitize the path from it to an observable point without violating constraints and/or the overall graph logic. Untested just means there is no existing Old Test that covers it. The only way to get Untested variations is by using the Run Old option.

Similarly, if we later found out that Tom is not going to the party, after having updated our tests, we can see what happens to that set of tests.



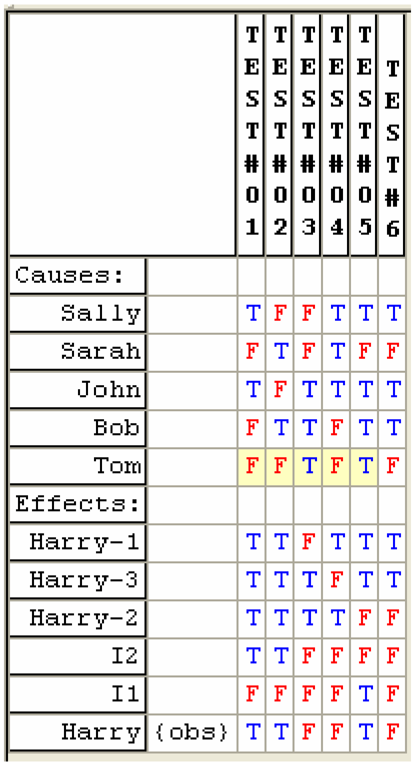
| | | TEST#1 | TEST#2 | TEST#3 | TEST#4 | TEST#5 | TEST#6 |
|---|---|---|---|---|---|---|---|
| Causes: | | | | | | | |
| Sally | | T | F | T | F | T | T |
| Sarah | | F | T | F | F | T | F |
| John | | T | F | T | T | T | T |
| Bob | | F | T | T | T | F | T |
| Effects: | | | | | | | |
| No-Harry-1 | | F | F | F | T | F | F |
| No-Harry-3 | | F | F | F | F | T | F |
| No-Harry-2 | | F | F | T | F | F | T |
| Harry | (obs) | T | T | F | F | F | F |
| Tom (Deleted) | | - | - | - | - | - | - |

**Harry Tests With Tom Deleted From Graph**

There will be no annotation on the test scripts that Tom has been deleted.

## 4.4 Supplementing Old Tests

You can then ask RBT to supplement the test cases to bring the coverage back up to 100%. From the RBTg interface choose Generate → Run Both. Again, you will be prompted for the test file that contains the existing tests. When this is done the new test definition matrix is:



| | | TEST#01 | TEST#02 | TEST#03 | TEST#04 | TEST#05 | TEST#6 |
|---|---|---|---|---|---|---|---|
| Causes: | | | | | | | |
| Sally | | T | F | F | T | T | T |
| Sarah | | F | T | F | T | F | F |
| John | | T | F | T | T | T | T |
| Bob | | F | T | T | F | T | T |
| Tom | | F | F | T | F | T | F |
| Effects: | | | | | | | |
| Harry-1 | | T | T | F | T | T | T |
| Harry-3 | | T | T | T | F | T | T |
| Harry-2 | | T | T | T | T | F | F |
| I2 | | T | T | F | F | F | F |
| I1 | | F | F | F | F | T | F |
| Harry | (obs) | T | T | F | F | T | F |

**Harry Tests – Run Both**

There is now a new Test #6. The matrix also still notes what must be added to the five old tests. In the Batch and Script Reports the new test is described and annotated as such:

New Test: TEST#6 -- Harry Goes To The Party - Tom Helps Out

Cause states:
Sally goes to the party
Sarah does not go to the party
John goes to the party
Bob goes to the party
Tom is not going to the party

Effect states:
Harry does not go to the party

## 4.5 Editing Old Tests

Once you save your tests you can edit them. You can change the test ID or even modify the list/state of the primary causes. To edit your tests you select:

Utilities → Preserve Tests → Open Old Tests

This lets RBT know which set of old tests you want to consider. Once the tests file has been selected you can Run-Old or Run-Both from the RBT interface. It also activates the test editor feature. You then open the test file you want to edit:

Utilities → Preserve Tests → Edit Old Tests

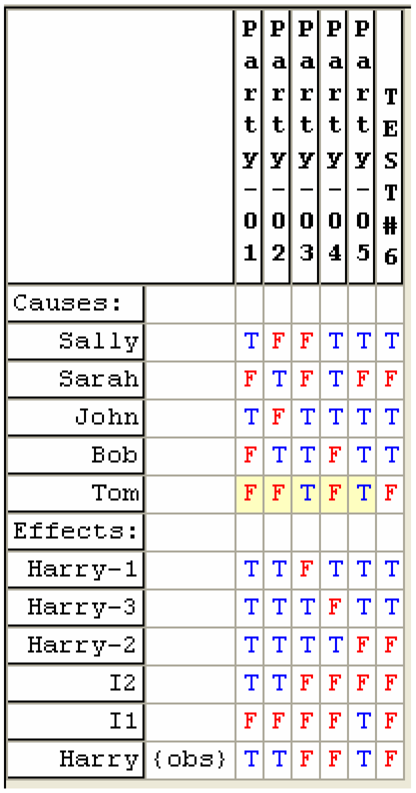
This brings up the test cases so you can edit them. Let us change the Test ID's from our original set of tests to something specific to the requirement. We change them to:

TESTS

Party-01 = Sally, not Sarah, John, not Bob.
Party-02 = not Sally, Sarah, not John, Bob.
Party-03 = not Sally, not Sarah, John, Bob.
Party-04 = Sally, Sarah, John, not Bob.
Party-05 = Sally, not Sarah, John, Bob.

Please note that word "TESTS" at the top is used by RBT to denote what type of data this is. When you Run Old or Run Both, RBT appends this information onto the input file. Do not change this key word, only the test case information.

We now run the updated graph (the one with Tom added) using this new version of the original tests. Selecting the Run Both option, the Test Definition Matrix is:



| | | Party-01 | Party-02 | Party-03 | Party-04 | Party-05 | TEST#6 |
|---|---|---|---|---|---|---|---|
| Causes: | | | | | | | |
| Sally | | T | F | F | T | T | T |
| Sarah | | F | T | F | T | F | F |
| John | | T | F | T | T | T | T |
| Bob | | F | T | T | F | T | T |
| Tom | | F | F | T | F | T | F |
| Effects: | | | | | | | |
| Harry-1 | | T | T | F | T | T | T |
| Harry-3 | | T | T | T | F | T | T |
| Harry-2 | | T | T | T | T | F | F |
| I2 | | T | T | F | F | F | F |
| I1 | | F | F | F | F | T | F |
| Harry | (obs) | T | T | F | F | T | F |

**Harry Tests – Run Both – New Test ID's**

(Note that the new test still follows RBT's simple Test ID naming convention. It is in our future enhancement list to allow the user to specify the test naming and numbering schema.)

When you edit your tests the changes are not reflected in any of the RBT reports until you run them. You need to do a Run Old or a Run Both to update the reports. If you are exporting the tests to another tool (e.g. Test Director) it is critical that you rerun prior to the export.

If you export the tests to one of test managers RBT exports the entire set of tests. We do not go into the test manager tool to figure what is new versus changed versus unchanged. We suggest you export the tests into a new file in the test manager. Then work in the test manager tool to copy over the new and changed tests as needed.

## *4.6 Telling RBT About Non-RBT Tests*

It is quite possible that you already have some test cases built for the function that you just graphed. For example, you have tests from a prior release of the application not tested via RBT. However, these tests represent a significant investment. You want to start with these tests as a base and have RBT only supplement them.

To tell RBT about existing tests you have two choices. You can use a simple text editor, define the tests using the format above, and save the file as a .CET file. Do not forget to include the key word "TESTS". Also note that each test definition ends in a period. If you use something like MS Word to do this you first need to save it as a .TXT file. Then change it to a .CET file.

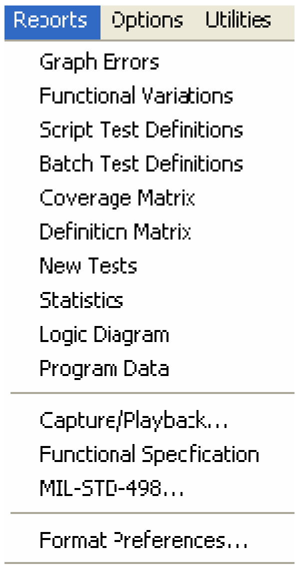
The second approach is to do a Run New of your graph. Save these tests to create a .CET file. Bring it up in the test editor. Define your existing tests while deleting the ones defined by RBT. You file will be in the right format and contain only the pre-existing tests.

After you have defined your existing tests, then use the Run Old option to evaluate how much they cover. Check the Coverage Matrix Report. This will quickly show you if there are any Untested variations. Also, you can see if you have any redundant tests. A variation covered by only one test is denoted with a "#". If a variation is in two or more tests it is marked with an "X". Any test designed by RBT has at least one variation that is not covered by another test. If you have tests that are all "X"'s, then there is some redundancy.

Using the information in the Coverage Matrix you can decide which tests to delete while still keeping the same level of test coverage. However, the "redundant" tests might be there because of design dependent or code dependent considerations. Therefore, take care before deleting tests from your existing test library.

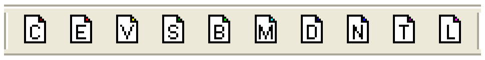
# 5. Reports

RBT generates a number of reports once you have generated your tests. You can access the set of reports by selecting the Reports button on the main menu. That will take you from RBTg over to RBT where all of the data is accessible. Once in RBT you can access the list of reports via the Report pull down menu:



**Reports Pull Down Menu**

Most of these reports are also accessible via buttons across the top of the screen:



**Reports Buttons**

The **Cause-Effect Graph** report [C] is the API version of the graph created in RBTg.

The **Graph Errors** report [E] identifies any problems with the graph. These may be just at the warning (W) level. They may also be at the severe (S) level in which case the graph did not compile into tests.

The **Functional Variations** report [V] lists all of the functional variations for the graph and their status – e.g. infeasible, untestable.

The **Script Test Definitions** report [S] is the set of tests presented as an interactive dialog of causes and effects.

The **Batch Test Definitions** report [B] is the same set of tests with all of the causes listed then all of the effects.

The **Coverage Matrix** [M] identifies which variations are included in which test cases.

The **Definition Matrix** [D] identifies the state of each of the nodes in each test case – i.e. true, false, or masked.

The **New Tests** report [N] shows what RBT will save if you ask it to remember your tests.

The **Test Statistics** report [T] (know internally as the Golly Gee Wiz report) gives you statistics about your graph and the tests.

The **Logic Diagram** [L] is another view of the graph drawn by RBT from the graph input. This is useful if you are entering your graphs via the API and not RBTg.

The **Program Data** report contains all of the information RBT knows about the graph and the tests. This data can then be exported and used as input to executable test generation tools.

The **Capture/Playback** report generates the tests in the format that various capture playback tools want as input. It only generates them as comment statements.

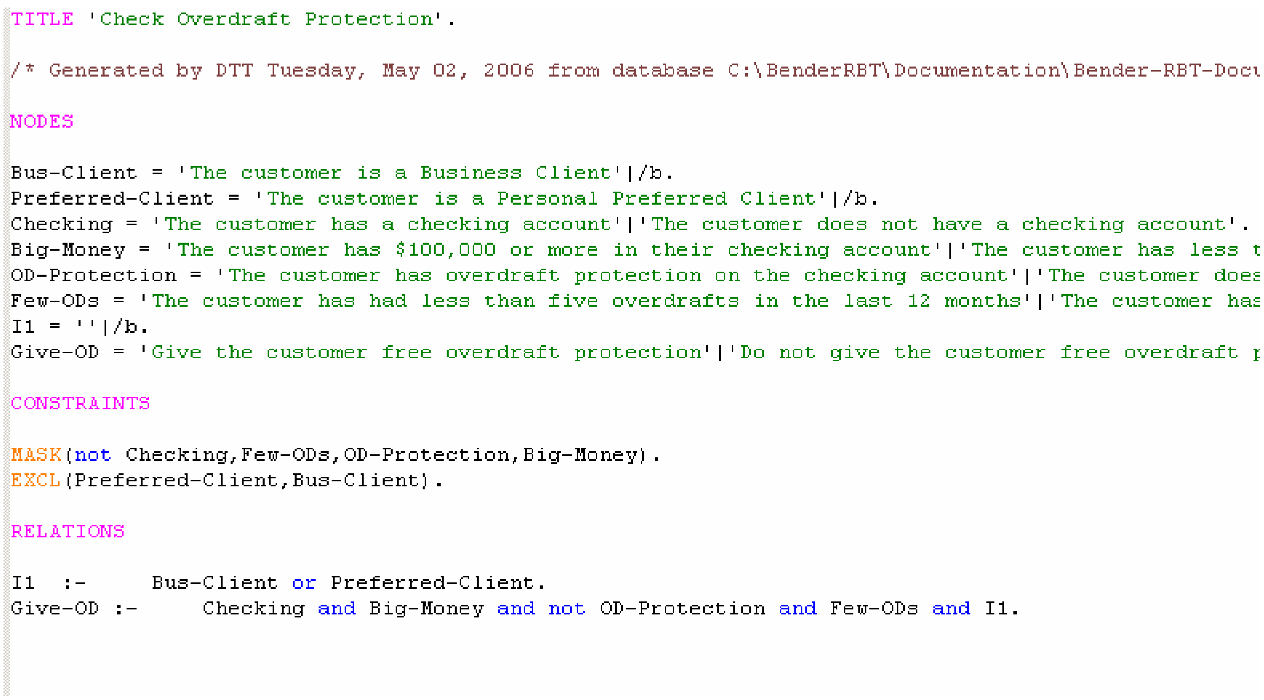
The **Functional Specification** report generates a requirements document from the graph information.

The **MIL-STD-498** generates the test descriptions in a format that some military projects prefer to use.

The **Format Preferences** allows you to tune what is displayed on the Script and Batch Test Case reports.

## *5.1 Cause-Effect Graph Report*

RBTg generates the Prolog syntax version of the input to the RBT test design engine. This is also the input format used by the API. Generally a user of RBT would never directly manipulate this information.



```
TITLE 'Check Overdraft Protection'.

/* Generated by DTT Tuesday, May 02, 2006 from database C:\BenderRBT\Documentation\Bender-RBT-Docu

NODES

Bus-Client = 'The customer is a Business Client'|/b.
Preferred-Client = 'The customer is a Personal Preferred Client'|/b.
Checking = 'The customer has a checking account'|'The customer does not have a checking account'.
Big-Money = 'The customer has $100,000 or more in their checking account'|'The customer has less t
OD-Protection = 'The customer has overdraft protection on the checking account'|'The customer does
Few-ODs = 'The customer has had less than five overdrafts in the last 12 months'|'The customer has
I1 = ''|/b.
Give-OD = 'Give the customer free overdraft protection'|'Do not give the customer free overdraft p

CONSTRAINTS

MASK(not Checking,Few-ODs,OD-Protection,Big-Money).
EXCL(Preferred-Client,Bus-Client).

RELATIONS

I1  :-     Bus-Client or Preferred-Client.
Give-OD :-     Checking and Big-Money and not OD-Protection and Few-ODs and I1.
```

**Cause-Effect Graph Report**

## 5.2 Graph Errors Report

The cause-effect graph file is parsed during the initial processing phase of designing tests. Any errors detected are listed in the Graph Errors report. Any errors which are not significant enough to necessitate correction and subsequent run processing, or other possible anomalies that the software may note, are reported as Warning messages and identified with a code starting with the letter 'W.' Any errors which will require some corrective action before further processing may continue are reported as Severe errors, and identified with a code starting with the letter 'S.'

The error messages are generally self-explanatory, but additional insight into the error conditions and suggested corrective actions are given in Chapter 10: Diagnostic and Error Messages.

```
Graph Errors

S37--Extraneous information in:
S..      Bus Client = 'The customer is a business client' | /b.
S31--Node name used in Relations statement not defined or invalid: Bus_Client
W02--Defined node not used in graph: Bus

Graph Entry  phase completed Return Code = 6
```

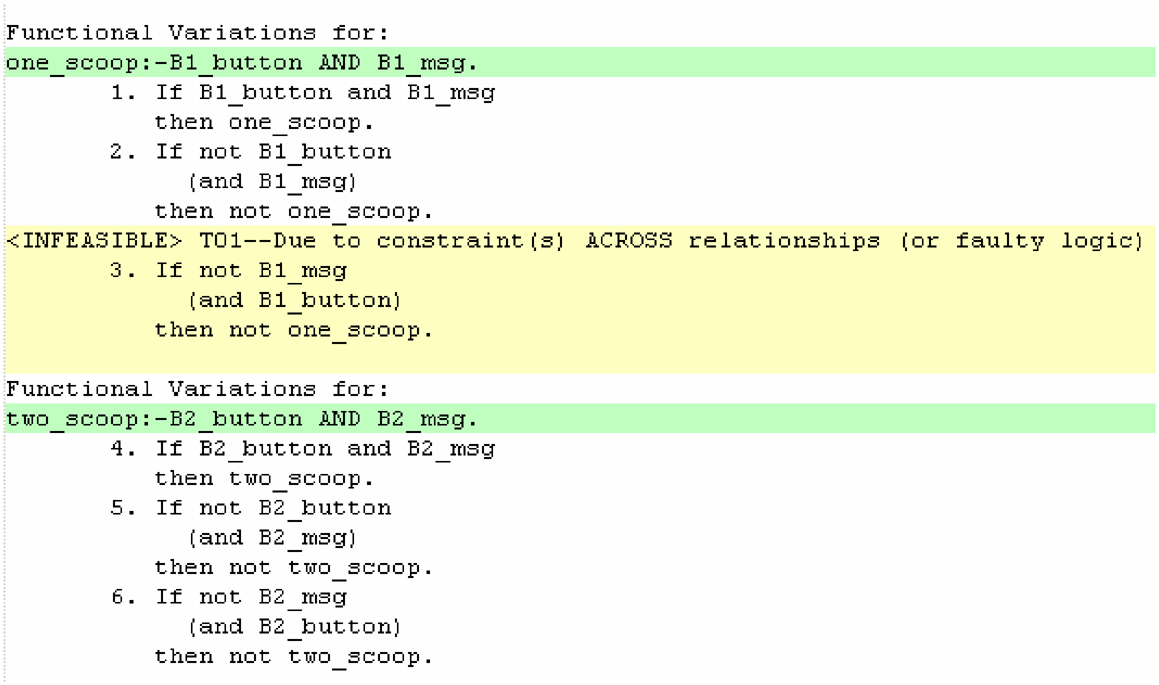
**Graph Errors Report**

## *5.3 Functional Variations Report*

The primitive variants of the graphed function are listed in the Functional Variation Report. For each variation the following information is included:

- A restatement of the Relations statement
- A serially-assigned variation number
- The cause(s) and their state(s)
- The effect and its state

If a node name is preceded by the word NOT, the node's state is false in the variation; otherwise, it is true.



```
Functional Variations for:
one_scoop:-B1_button AND B1_msg.
       1. If B1_button and B1_msg
          then one_scoop.
       2. If not B1_button
            (and B1_msg)
          then not one_scoop.
<INFEASIBLE> T01--Due to constraint(s) ACROSS relationships (or faulty logic)
       3. If not B1_msg
            (and B1_button)
          then not one_scoop.

Functional Variations for:
two_scoop:-B2_button AND B2_msg.
       4. If B2_button and B2_msg
          then two_scoop.
       5. If not B2_button
            (and B2_msg)
          then not two_scoop.
       6. If not B2_msg
            (and B2_button)
          then not two_scoop.
```

**Functional Variations Report**

The primary sensitizing condition(s) for each functional variation are listed first, after the beginning word "If ..." followed by the non-primary sensitizing condition(s), which is enclosed within a pair of parentheses in the functional variation definition.

If a variation proves to be infeasible due to constraints and/or the overall logic, this is indicated by an infeasible diagnostic message preceding the variation's definition.

If a variation proves to be Untestable due to the system's inability to observe either the true or false state of its effect node, this is indicated by an untestable diagnostic message preceding the variation's definition.

If the user selected the Run-Old option to evaluate an existing set of tests, then some variations might be identified as "Untested". This means that the current test library does not cover them and the tests need to be supplemented to complete the coverage.

Additional diagnostic messages are inserted by BenderRBT when deemed appropriate. For example, if all of the variations that lead to an effect state being true (or the variations where an effect is false) have been flagged as infeasible, then it is possible that the graph statements are illogically defined. A diagnostic message reports this and other conditions that you should review and take action as deemed appropriate.

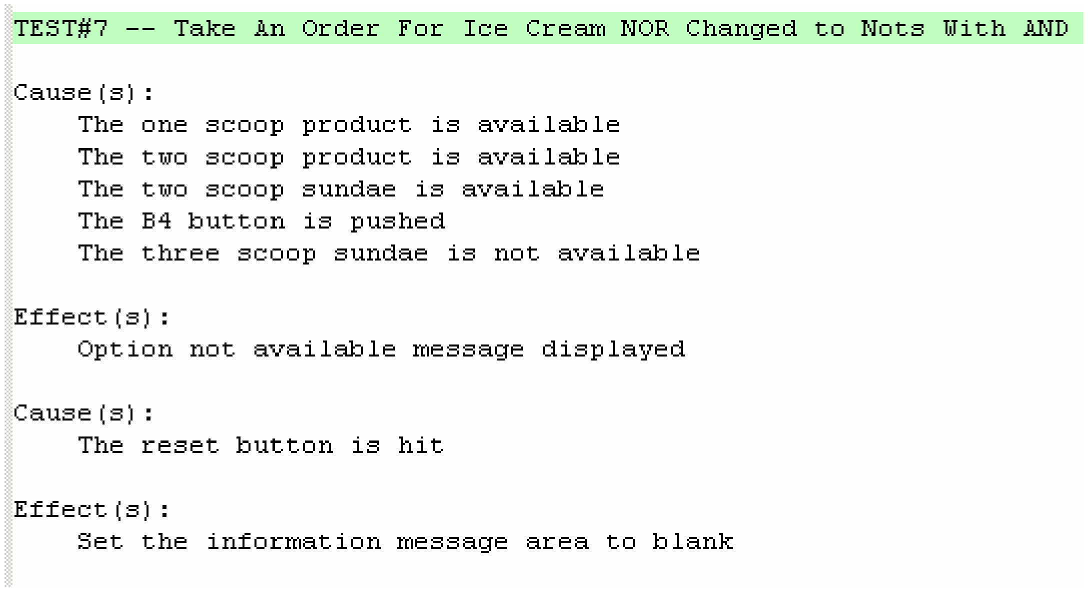
See Chapter 10: Diagnostic and Error Messages for a review of all of the diagnostic messages that may appear.

## 5.4 Script Test Case Definitions Report

The Script Test Case Definitions report is for interactive systems. A test would have some causes then some effects followed by more causes and effects as the dialog continues. The Script Tests lists causes and effects in their logically declared sequence (i.e., they appear in groups of related causes followed by their associated effects for each of the test cases generated).

A serially-assigned test case number is created beginning with the literal TEST# and displayed whenever an old test case definition has not been specified. When an old test case definition is being listed, its defined test case name is displayed.

If the option to include node names has been specified for this report (see the Reports → Format Preferences), then throughout the test case definitions, a false node state is indicated by a not preceding the node name; otherwise, the absence of a not when the node name is present indicates the true state of the node.

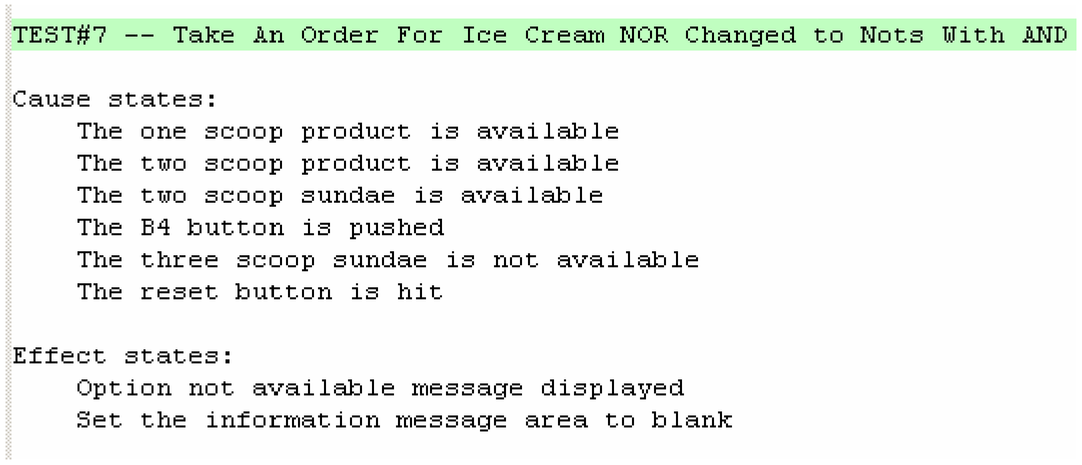


```
TEST#7 -- Take An Order For Ice Cream NOR Changed to Nots With AND

Cause(s):
    The one scoop product is available
    The two scoop product is available
    The two scoop sundae is available
    The B4 button is pushed
    The three scoop sundae is not available

Effect(s):
    Option not available message displayed

Cause(s):
    The reset button is hit

Effect(s):
    Set the information message area to blank
```

**Script Test Case Report**

## 5.5 Batch Test Case Definition Report

For tests that are not interactive you would use the Batch Test Case Definition report. This lists all of the causes followed by all of the effects for each of the test cases generated.

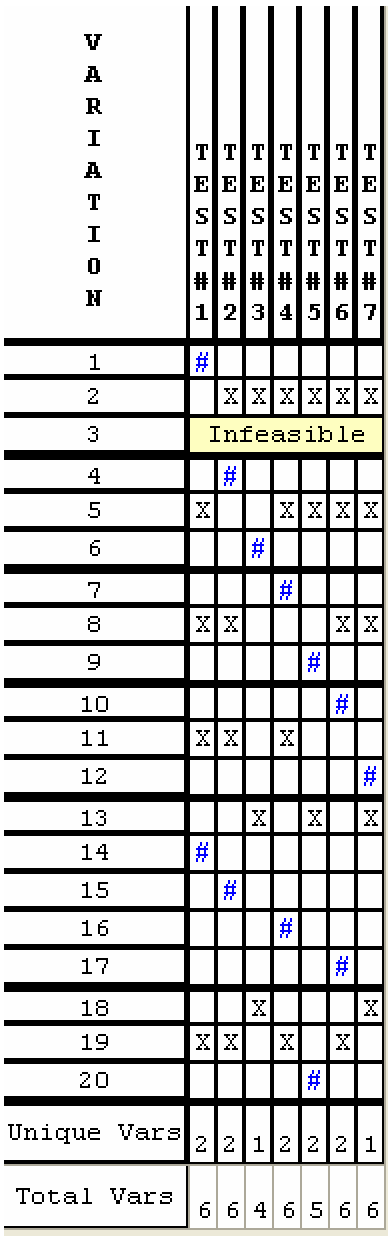


```
TEST#7 -- Take An Order For Ice Cream NOR Changed to Nots With AND

Cause states:
    The one scoop product is available
    The two scoop product is available
    The two scoop sundae is available
    The B4 button is pushed
    The three scoop sundae is not available
    The reset button is hit

Effect states:
    Option not available message displayed
    Set the information message area to blank
```

**Batch Case Definition Report**

## 5.6 Coverage Matrix

The functional variation coverage achieved by each test is analyzed and presented in the test case versus functional variation Coverage Matrix. The variations and tests are identified by their assigned numbers and test names. An ‘X’ in an intersecting row and column on the matrix indicates the validation of the functional variation within the named test case. A pound sign (#) in an intersecting row and column on the matrix indicates that this is the only test case (within the suite of test cases defined) in which the functional variation occurs.



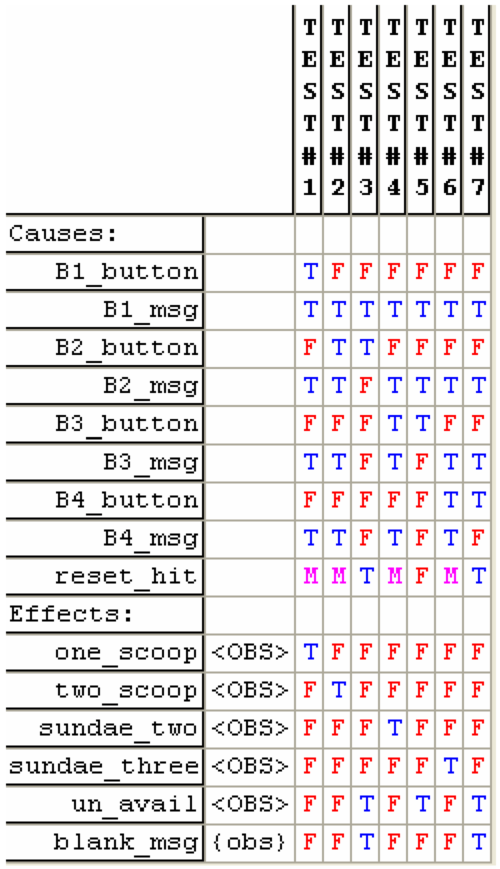
| VARIATION | TEST #1 | TEST #2 | TEST #3 | TEST #4 | TEST #5 | TEST #6 | TEST #7 |
|---|---|---|---|---|---|---|---|
| 1 | # | | | | | | |
| 2 | | X | X | X | X | X | X |
| 3 | Infeasible | | | | | | |
| 4 | | # | | | | | |
| 5 | X | | | X | X | X | X |
| 6 | | | # | | | | |
| 7 | | | | # | | | |
| 8 | X | X | | | | X | X |
| 9 | | | | | # | | |
| 10 | | | | | | # | |
| 11 | X | X | | X | | | |
| 12 | | | | | | | # |
| 13 | | | X | | X | | X |
| 14 | # | | | | | | |
| 15 | | # | | | | | |
| 16 | | | | # | | | |
| 17 | | | | | | # | |
| 18 | | | X | | | | X |
| 19 | X | X | | X | | X | |
| 20 | | | | | # | | |
| Unique Vars | 2 | 2 | 1 | 2 | 2 | 2 | 1 |
| Total Vars | 6 | 6 | 4 | 6 | 5 | 6 | 6 |

**Coverage Matrix**

Functional variations that are listed in this report but are not covered by any of the test cases generated are flagged as either infeasible, untestable, or untesed. These variations are highlighted by a unique color. The sets of functional variations generated for each relations statement are separated by a bold horizontal bar on the displayed matrix.

## 5.7 Definition Matrix

The Definition matrix is a compact depiction of the state of the causes and effects for each test case. For each test the node states should be either true [T], false [F], or masked [M]. Node names are listed in rows and test case names in columns. In the upper half of the matrix, each of the graph's primary causes is listed, followed by their respective node states in each of the test cases. In the lower half of the Definition Matrix, each effect node name and its state is listed. Any observable or forced-observable intermediate node is denoted by the appearance of the legend OBS or FOBS respectively. All primary effect node names (which are implicitly treated as observable) are designated OBS.



| | | TEST#1 | TEST#2 | TEST#3 | TEST#4 | TEST#5 | TEST#6 | TEST#7 |
|---|---|---|---|---|---|---|---|---|
| Causes: | | | | | | | | |
| B1_button | | T | F | F | F | F | F | F |
| B1_msg | | T | T | T | T | T | T | T |
| B2_button | | F | T | T | F | F | F | F |
| B2_msg | | T | T | F | T | T | T | T |
| B3_button | | F | F | F | T | T | F | F |
| B3_msg | | T | T | F | T | F | T | T |
| B4_button | | F | F | F | F | F | T | T |
| B4_msg | | T | T | F | T | F | T | F |
| reset_hit | | M | M | T | M | F | M | T |
| Effects: | | | | | | | | |
| one_scoop | <OBS> | T | F | F | F | F | F | F |
| two_scoop | <OBS> | F | T | F | F | F | F | F |
| sundae_two | <OBS> | F | F | F | T | F | F | F |
| sundae_three | <OBS> | F | F | F | F | F | T | F |
| un_avail | <OBS> | F | F | T | F | T | F | T |
| blank_msg | {obs} | F | F | T | F | F | F | T |

**Test Definition Matrix**

If there is a small "t" or "f" in a cell it means that RBT filled this in to complete the test definition. These nodes are not the primary objects being tested. In cases where it is non-trivial to verify test results, the tester might make a judgment call not to explicitly verify the results for these nodes when the test is executed. The general guideline, however, is to verify all test results.

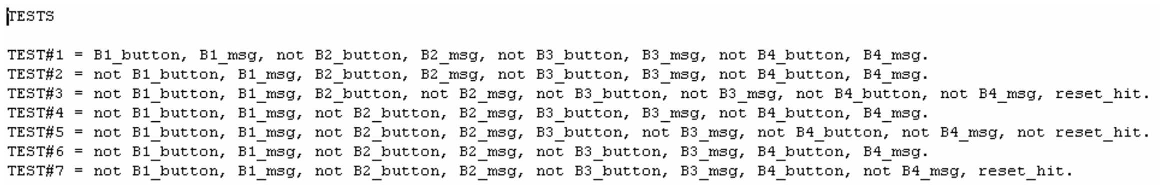
Some masked nodes will have a capital "M" while others might have a small "m". The capital "M" denotes that the node was explicitly masked for this test in the graph. The small "m" denotes that RBT deduced that the node was masked by extrapolation.

If you have an "I" in the matrix, it means that the node state was indeterminate. Indeterminate results are always an error – either in the graph or in the original logic in the requirement. The most common cause of them is an incomplete Mask constraint. A capital "I" denotes that this is the point where the effect became indeterminate. A small "i" denotes that this node is indeterminate by extrapolation. Focus on the points in the graph where the capital "I" appeared to debug the problem.

In some cases you might have a space in a cell. This means that RBT could not fill in anything and still be consistent with the logic of the graph. RBT uses functional variations as the building blocks for defining the tests. Sometimes the test would imply that two or more causes in an OR would have to be true. Since an OR relation would only generate variations with at most one true, RBT would not be able to complete the test definition. (Note: this will be addressed in the next generation of the test design engine.) In such cases the tester needs to complete the test definition manually.

## 5.8 New TESTS Report

As discussed in the chapter on managing tests, RBT remembers the tests it has previously designed via the Tests file. This report shows those tests.



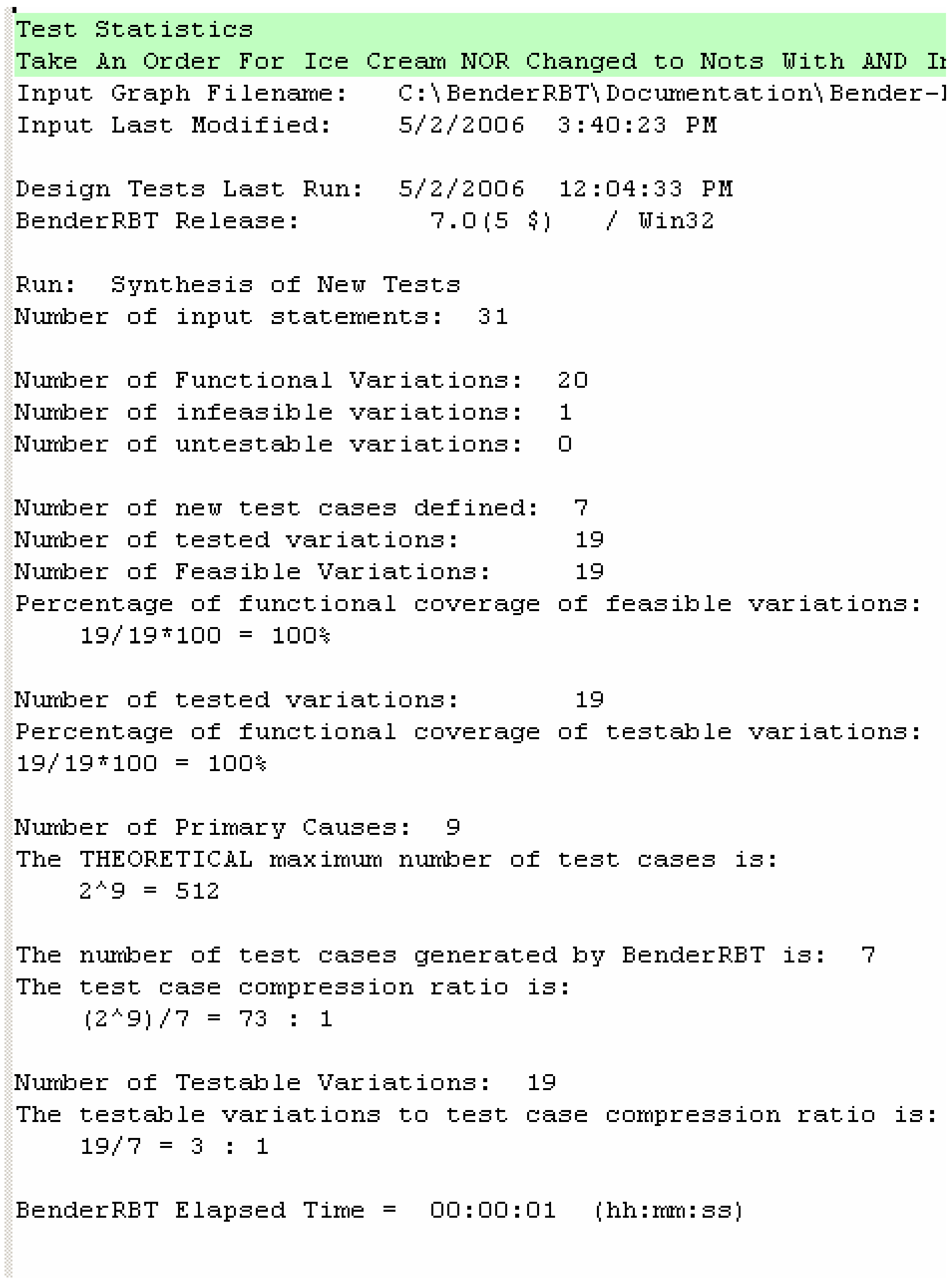
```
TESTS

TEST#1 = B1_button, B1_msg, not B2_button, B2_msg, not B3_button, B3_msg, not B4_button, B4_msg.
TEST#2 = not B1_button, B1_msg, B2_button, B2_msg, not B3_button, B3_msg, not B4_button, B4_msg.
TEST#3 = not B1_button, B1_msg, B2_button, not B2_msg, not B3_button, not B3_msg, not B4_button, not B4_msg, reset_hit.
TEST#4 = not B1_button, B1_msg, not B2_button, B2_msg, B3_button, B3_msg, not B4_button, B4_msg.
TEST#5 = not B1_button, B1_msg, not B2_button, B2_msg, B3_button, not B3_msg, not B4_button, not B4_msg, not reset_hit.
TEST#6 = not B1_button, B1_msg, not B2_button, B2_msg, not B3_button, B3_msg, B4_button, B4_msg.
TEST#7 = not B1_button, B1_msg, not B2_button, B2_msg, not B3_button, B3_msg, B4_button, not B4_msg, reset_hit.
```

**New Tests Report**

Note that only the causes for each test are defined. In using this test information RBT will deduce what the effect states will be based on the graph.

## 5.9 Test Statistics Report

The Test Statistics report summarizes data about the graph including how many variations, the number of tests, the number of possible tests, the run time, etc.



```
Test Statistics
Take An Order For Ice Cream NOR Changed to Nots With AND In
Input Graph Filename:   C:\BenderRBT\Documentation\Bender-
Input Last Modified:    5/2/2006  3:40:23 PM

Design Tests Last Run:  5/2/2006  12:04:33 PM
BenderRBT Release:        7.0(5 $)   / Win32

Run:  Synthesis of New Tests
Number of input statements:  31

Number of Functional Variations:  20
Number of infeasible variations:  1
Number of untestable variations:  0

Number of new test cases defined:  7
Number of tested variations:       19
Number of Feasible Variations:     19
Percentage of functional coverage of feasible variations:
    19/19*100 = 100%

Number of tested variations:       19
Percentage of functional coverage of testable variations:
19/19*100 = 100%

Number of Primary Causes:  9
The THEORETICAL maximum number of test cases is:
    2^9 = 512

The number of test cases generated by BenderRBT is:  7
The test case compression ratio is:
    (2^9)/7 = 73 : 1

Number of Testable Variations:  19
The testable variations to test case compression ratio is:
    19/7 = 3 : 1

BenderRBT Elapsed Time =  00:00:01  (hh:mm:ss)
```

**Test Statistics Report**

For any of the calculations of the percentage of functional variation coverage achieved by the test library (both Old and New tests), the numerator used is the actual number of tested variations present in the test cases. The denominator used when computing the percentage of coverage of feasible variations is the total number of functional variations generated less any infeasible variations. The denominator used when computing the percentage of coverage of testable variations is the total number of functional variations generated less the sum of any infeasible and untestable variations.

The reason we call this the "Golly Gee Wiz" report is because of some of the statistics that come from larger graphs. The following statistics are from the graph testing a function embedded in a car:

Number of Primary Causes: 142

The THEORETICAL maximum number of test cases is:
2^142 = 5,575,186,299,632,655,800,000,000,000,000,000,000,000,000

The number of test cases generated by BenderRBT is: 137

The test case compression ratio is:
(2^142)/137 = 40,694,790,508,267,559,000,000,000,000,000,000,000,000 : 1

BenderRBT Elapsed Time = 00:01:28 hh:mm:ss)

(This is the current record. If you come up with one bigger, please send it to me at rbender@BenderRBT.com.)

## *5.10 Logic Diagram Report*

The RBT engine can generate a picture of the graph via the Logic Diagram. This is for use in those instances where the data was entered via the API instead of RBTg. The other option is to import the .ceg file into RBTg as discussed earlier.

Nodes are placed on the diagram under the control of internal programmed logic, with an attempt made to minimize any occurrence of unrelated vector lines crossing over other relational operator vector junctions. There is no provision for the user to modify the placement of nodes.

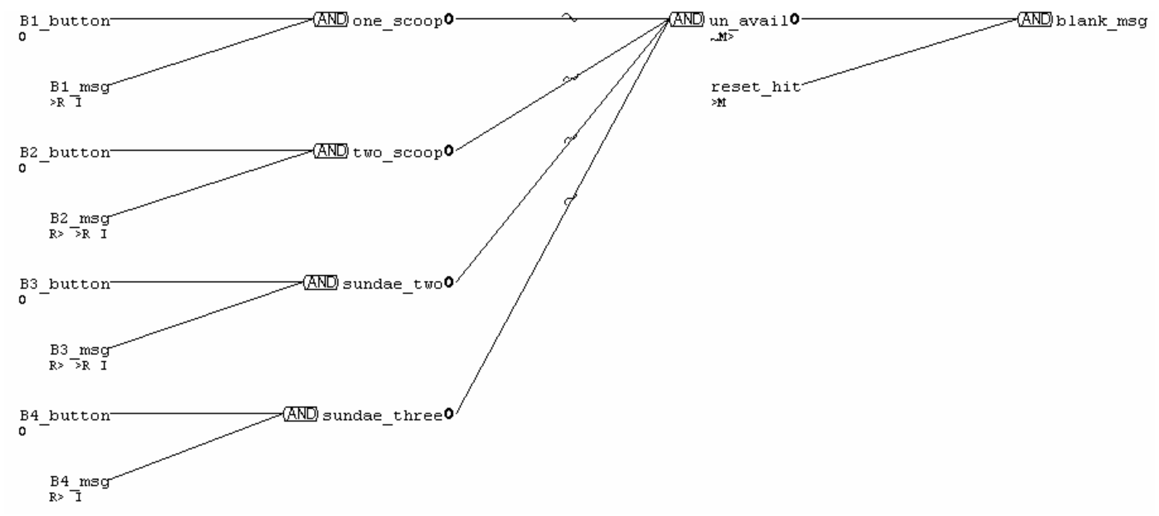
Any one of three symbol sets can be used to graphically depict the relational operators on the Logic Diagram.

Any nodes named in a constraints statement are annotated below the node name on the Logic Diagram using the following abbreviations:

| | |
|---|---|
| ~ | the false state of the node is specified in the Constraint |
| M> | the node is the subject of a Mask Constraint |
| >M | the node is an object of a Mask Constraint |

| | |
|---|---|
| R> | the node is the subject of a Requires Constraint |
| >R | the node is an object of a Requires Constraint |
| E | the node is named in an Exclusive Constraint |
| I | the node is named in an Inclusive Constraint |
| O | the node is named in a One-and-only-one Constraint |
| A | the node is named in an Anchor Constraint |

For example, the annotation ~M> indicates that the false state of the node is the subject of one or more Mask Constraint(s).



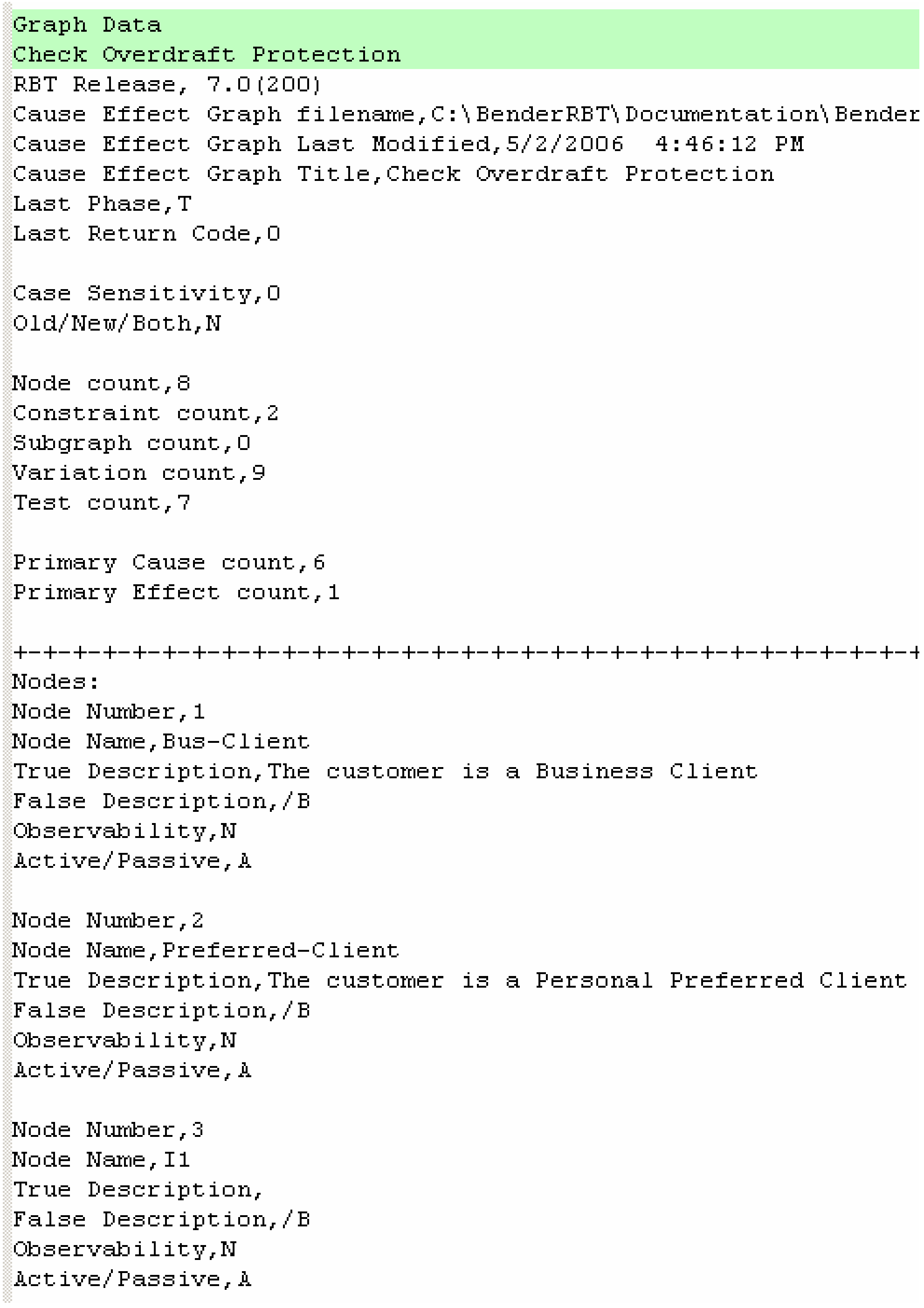
**Logic Diagram Report**

Any intermediate node which has been declared to be Observable is noted on the Logic Diagram with a superscript bold **o** symbol following the node name. Any intermediate node which has been declared as Forced Observable is noted on the diagram with a superscript ◉ symbol following the node name.

When any Logic Diagram is printed and multiple pages of output are produced, the notation "Row x Column y" is printed at the bottom of each page as an aid in laying out the segments of the diagram.

## *5.11 Program Data Report*

The Program Data Report contains all of the key information about the nodes, constraints, functional variations, and tests for a given graph. The intent of this report is for use in interfacing to tools, homegrown or off the shelf, used to create executable tests. To use this file you would probably export it as a csv file (comma delimited file). This could then be brought up in Excel or imported directly into the tool.

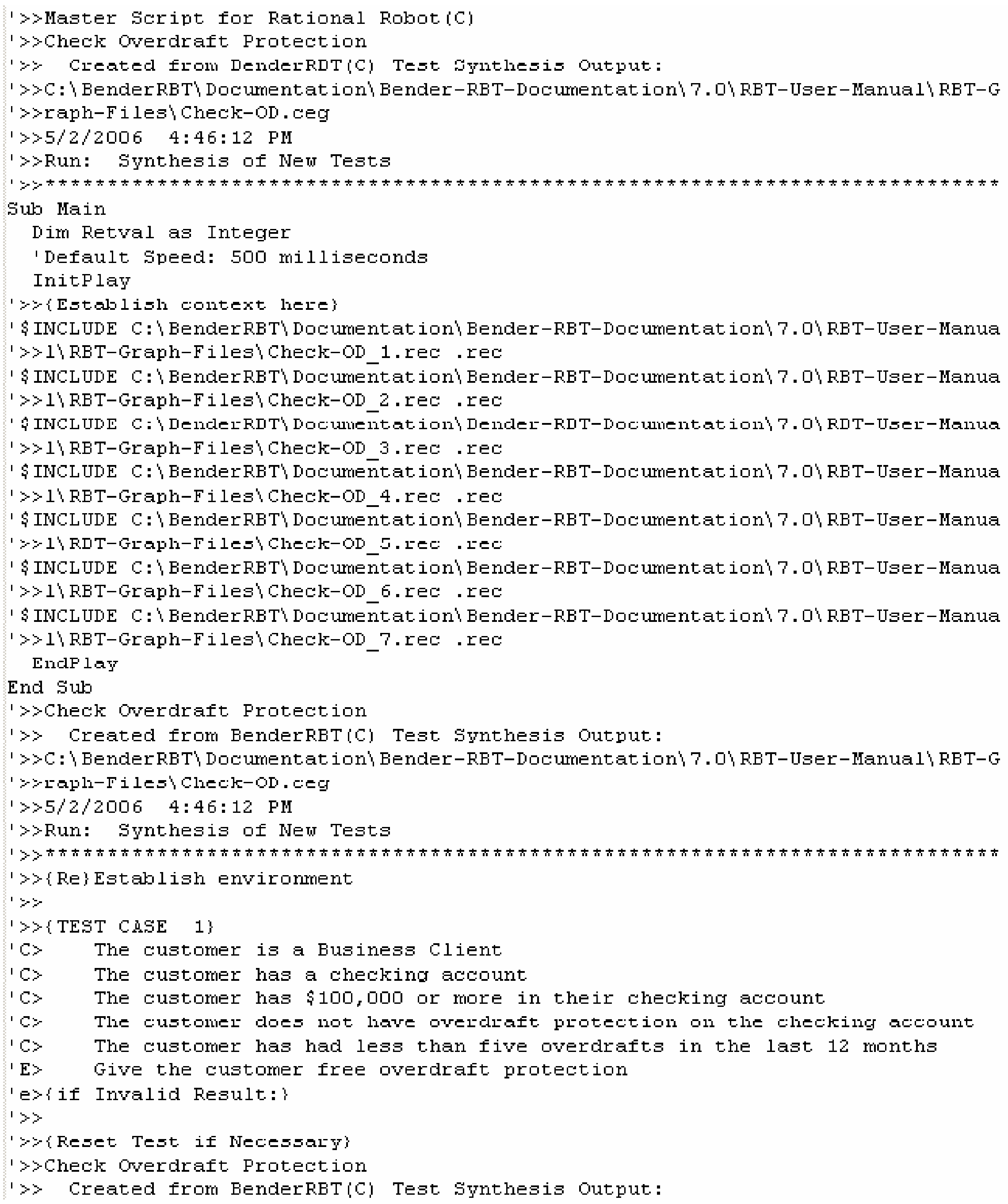


```
Graph Data
Check Overdraft Protection
RBT Release, 7.0(200)
Cause Effect Graph filename,C:\BenderRBT\Documentation\Bender
Cause Effect Graph Last Modified,5/2/2006  4:46:12 PM
Cause Effect Graph Title,Check Overdraft Protection
Last Phase,T
Last Return Code,0

Case Sensitivity,0
Old/New/Both,N

Node count,8
Constraint count,2
Subgraph count,0
Variation count,9
Test count,7

Primary Cause count,6
Primary Effect count,1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Nodes:
Node Number,1
Node Name,Bus-Client
True Description,The customer is a Business Client
False Description,/B
Observability,N
Active/Passive,A

Node Number,2
Node Name,Preferred-Client
True Description,The customer is a Personal Preferred Client
False Description,/B
Observability,N
Active/Passive,A

Node Number,3
Node Name,I1
True Description,
False Description,/B
Observability,N
Active/Passive,A
```

**Program Data Report**

## 5.12 Capture/Playback Report

RBT does not generate executable tests unless used in conjunction with the DTT add-on. It can, however, export the tests to other off-the-shelf playback tools. It does this by putting the tests in the form of comment statement using the format expected by the target tool. It then creates a file for each test and an include file for the set of tests. This gives the user some degree of self documenting tests.

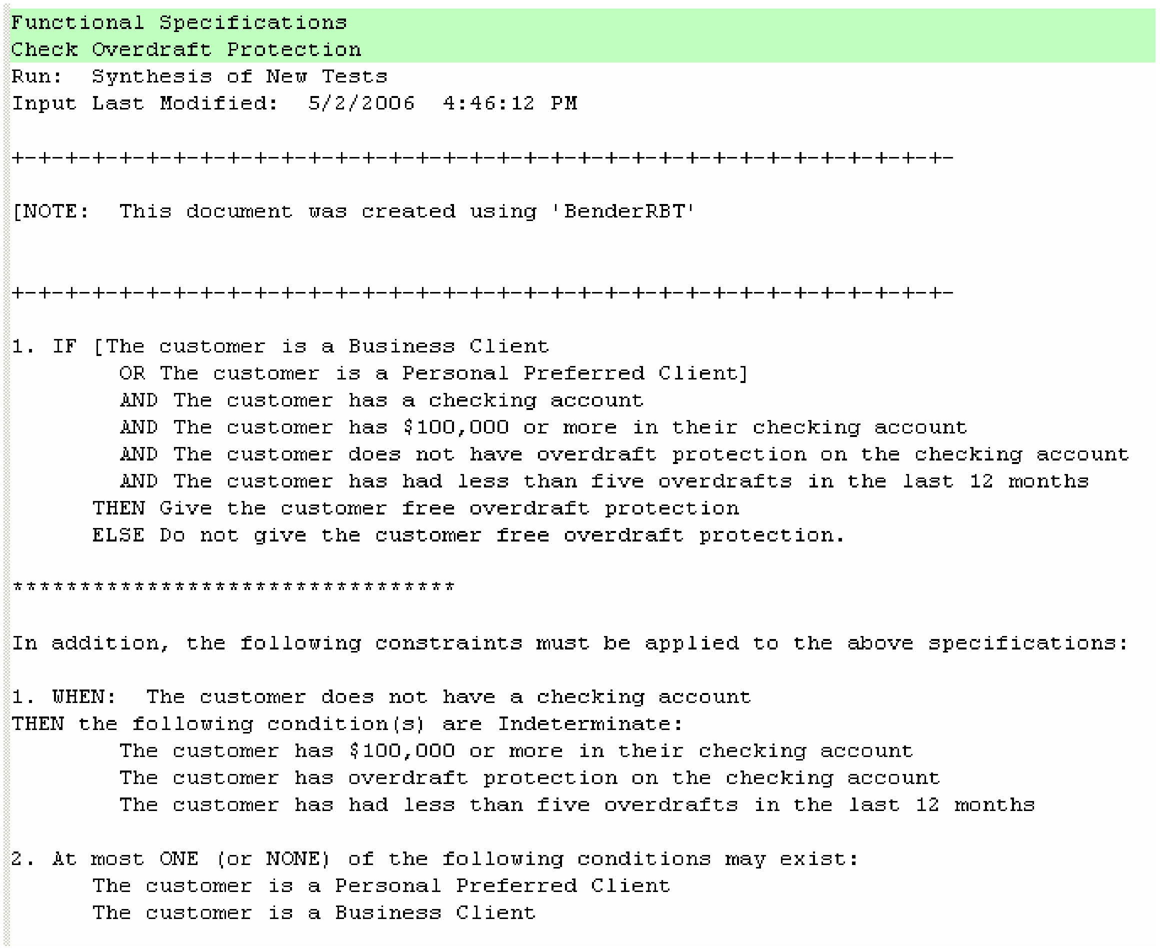


```
'>>Master Script for Rational Robot(C)
'>>Check Overdraft Protection
'>>  Created from BenderRBT(C) Test Synthesis Output:
'>>C:\BenderRBT\Documentation\Bender-RBT-Documentation\7.0\RBT-User-Manual\RBT-G
'>>raph-Files\Check-OD.ceg
'>>5/2/2006  4:46:12 PM
'>>Run:  Synthesis of New Tests
'>>**************************************************************************
Sub Main
  Dim Retval as Integer
  'Default Speed: 500 milliseconds
  InitPlay
'>>{Establish context here}
'$INCLUDE C:\BenderRBT\Documentation\Bender-RBT-Documentation\7.0\RBT-User-Manua
'>>l\RBT-Graph-Files\Check-OD_1.rec .rec
'$INCLUDE C:\BenderRBT\Documentation\Bender-RBT-Documentation\7.0\RBT-User-Manua
'>>l\RBT-Graph-Files\Check-OD_2.rec .rec
'$INCLUDE C:\BenderRBT\Documentation\Bender-RBT-Documentation\7.0\RBT-User-Manua
'>>l\RBT-Graph-Files\Check-OD_3.rec .rec
'$INCLUDE C:\BenderRBT\Documentation\Bender-RBT-Documentation\7.0\RBT-User-Manua
'>>l\RBT-Graph-Files\Check-OD_4.rec .rec
'$INCLUDE C:\BenderRBT\Documentation\Bender-RBT-Documentation\7.0\RBT-User-Manua
'>>l\RBT-Graph-Files\Check-OD_5.rec .rec
'$INCLUDE C:\BenderRBT\Documentation\Bender-RBT-Documentation\7.0\RBT-User-Manua
'>>l\RBT-Graph-Files\Check-OD_6.rec .rec
'$INCLUDE C:\BenderRBT\Documentation\Bender-RBT-Documentation\7.0\RBT-User-Manua
'>>l\RBT-Graph-Files\Check-OD_7.rec .rec
  EndPlay
End Sub
'>>Check Overdraft Protection
'>>  Created from BenderRBT(C) Test Synthesis Output:
'>>C:\BenderRBT\Documentation\Bender-RBT-Documentation\7.0\RBT-User-Manual\RBT-G
'>>raph-Files\Check-OD.ceg
'>>5/2/2006  4:46:12 PM
'>>Run:  Synthesis of New Tests
'>>**************************************************************************
'>>{Re}Establish environment
'>>
'>>{TEST CASE  1}
'C>    The customer is a Business Client
'C>    The customer has a checking account
'C>    The customer has $100,000 or more in their checking account
'C>    The customer does not have overdraft protection on the checking account
'C>    The customer has had less than five overdrafts in the last 12 months
'E>    Give the customer free overdraft protection
'e>{if Invalid Result:}
'>>
'>>{Reset Test if Necessary}
'>>Check Overdraft Protection
'>>  Created from BenderRBT(C) Test Synthesis Output:
```

**Capture/Playback Report**

## *5.13 Functional Specification Report*

It is incredibly rare that a tester will ever see a detailed, unambiguous specification of a function's rules. Specifications are, unfortunately, not always updated after the ambiguity reviews and graphing process have identified issues. However, as long as the tester received answers to their questions, they can create a graph that accurately reflects the rules. What RBT does is sort the information another way to generate an "as built" Functional Specification from the graph information. The tester can then give this to the analysts/developers as an add value benefit of the overall RBT process.



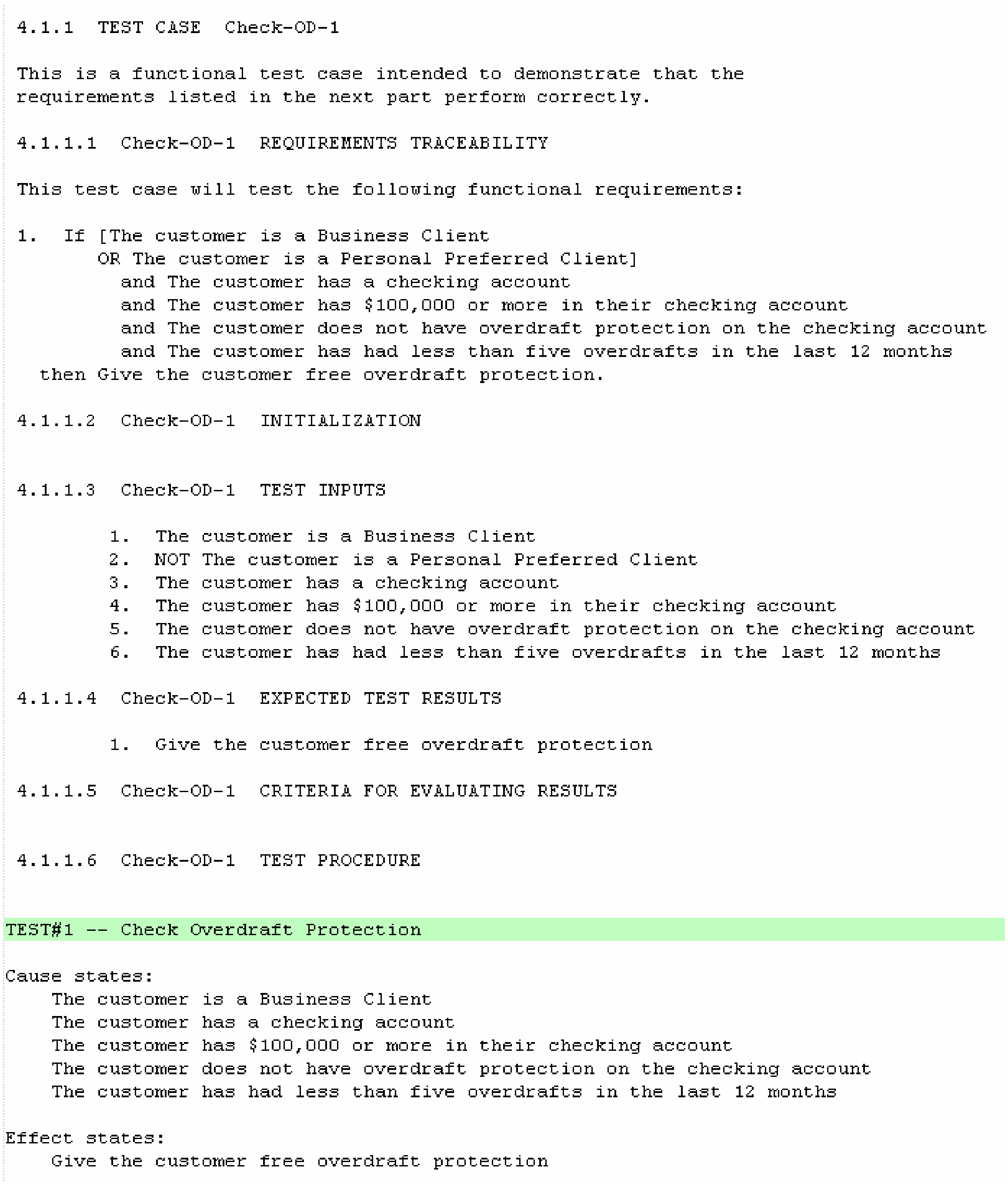
```
Functional Specifications
Check Overdraft Protection
Run:  Synthesis of New Tests
Input Last Modified:  5/2/2006  4:46:12 PM

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

[NOTE:  This document was created using 'BenderRBT'


+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

1. IF [The customer is a Business Client
        OR The customer is a Personal Preferred Client]
        AND The customer has a checking account
        AND The customer has $100,000 or more in their checking account
        AND The customer does not have overdraft protection on the checking account
        AND The customer has had less than five overdrafts in the last 12 months
      THEN Give the customer free overdraft protection
      ELSE Do not give the customer free overdraft protection.

*********************************

In addition, the following constraints must be applied to the above specifications:

1. WHEN:  The customer does not have a checking account
THEN the following condition(s) are Indeterminate:
        The customer has $100,000 or more in their checking account
        The customer has overdraft protection on the checking account
        The customer has had less than five overdrafts in the last 12 months

2. At most ONE (or NONE) of the following conditions may exist:
      The customer is a Personal Preferred Client
      The customer is a Business Client
```

**Functional Specification Report**

## *5.14 MIL-STD-498*

The United States Department of Defense had a series of standards for the overall development process down to the format for test cases. At one time there was 2167A. This was superseded by 498. Currently most military projects follow the IEEE standards.

RBT's standard reports conform to that guideline. The MIL-STD-498 standard for specifying tests combines the portion of the specification being tested with the test description itself. While this format is a bit of a legacy item we decided to keep it as an option.

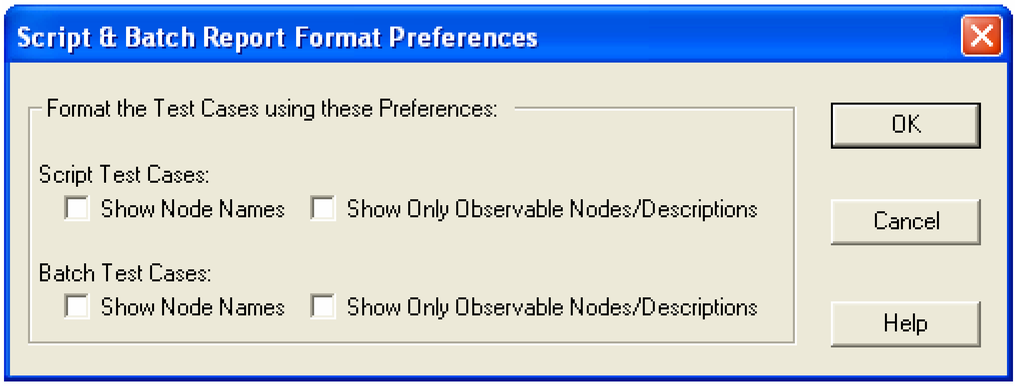


```
 4.1.1  TEST CASE  Check-OD-1

 This is a functional test case intended to demonstrate that the
 requirements listed in the next part perform correctly.

 4.1.1.1  Check-OD-1  REQUIREMENTS TRACEABILITY

 This test case will test the following functional requirements:

 1.  If [The customer is a Business Client
        OR The customer is a Personal Preferred Client]
          and The customer has a checking account
          and The customer has $100,000 or more in their checking account
          and The customer does not have overdraft protection on the checking account
          and The customer has had less than five overdrafts in the last 12 months
   then Give the customer free overdraft protection.

 4.1.1.2  Check-OD-1  INITIALIZATION


 4.1.1.3  Check-OD-1  TEST INPUTS

         1.  The customer is a Business Client
         2.  NOT The customer is a Personal Preferred Client
         3.  The customer has a checking account
         4.  The customer has $100,000 or more in their checking account
         5.  The customer does not have overdraft protection on the checking account
         6.  The customer has had less than five overdrafts in the last 12 months

 4.1.1.4  Check-OD-1  EXPECTED TEST RESULTS

         1.  Give the customer free overdraft protection

 4.1.1.5  Check-OD-1  CRITERIA FOR EVALUATING RESULTS


 4.1.1.6  Check-OD-1  TEST PROCEDURE


TEST#1 -- Check Overdraft Protection

Cause states:
    The customer is a Business Client
    The customer has a checking account
    The customer has $100,000 or more in their checking account
    The customer does not have overdraft protection on the checking account
    The customer has had less than five overdrafts in the last 12 months

Effect states:
    Give the customer free overdraft protection
```

**MIL-STD-498 Report**

## 5.15 Format Preferences

RBT allows you some flexibility as to what to display in both the Script and Batch form of the test cases. Selecting Reports → Format Preferences will bring up the following dialog:


Script & Batch Report Format Preferences

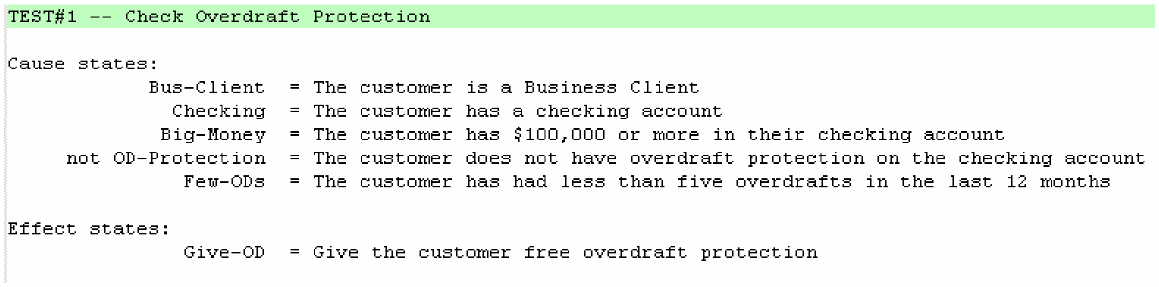
Format the Test Cases using these Preferences:

Script Test Cases:
☐ Show Node Names ☐ Show Only Observable Nodes/Descriptions

Batch Test Cases:
☐ Show Node Names ☐ Show Only Observable Nodes/Descriptions

**Format Preferences**

Showing the node names in the test descriptions can help you debug a graph. It makes it easier to go back and forth between the tests and the graph. For example, turning on the "Show Node Names" option would result in the following:


```
TEST#1 -- Check Overdraft Protection

Cause states:
            Bus-Client  = The customer is a Business Client
              Checking  = The customer has a checking account
             Big-Money  = The customer has $100,000 or more in their checking account
     not OD-Protection  = The customer does not have overdraft protection on the checking account
               Few-ODs  = The customer has had less than five overdrafts in the last 12 months

Effect states:
               Give-OD  = Give the customer free overdraft protection
```

**Batch Tests With Node Names**

## 5.16 Exporting The RBT Reports

Most of the RBT reports can be exported. The following reports are exported as text files (the file extension appears after the report name):

Graph Errors (.ge)
Functional Variations (.fv)
Script Test Definitions (.st)
Batch Test Definitions (.bt)

New Tests (.cet)
Test Statistics (.ggw)
Capture/Playback reports (various extensions – one for each tool)
Functional Specification (.frs)
MIL-STD-498 (.dod)

The following reports are exported as comma delimited files with the extension .csv:

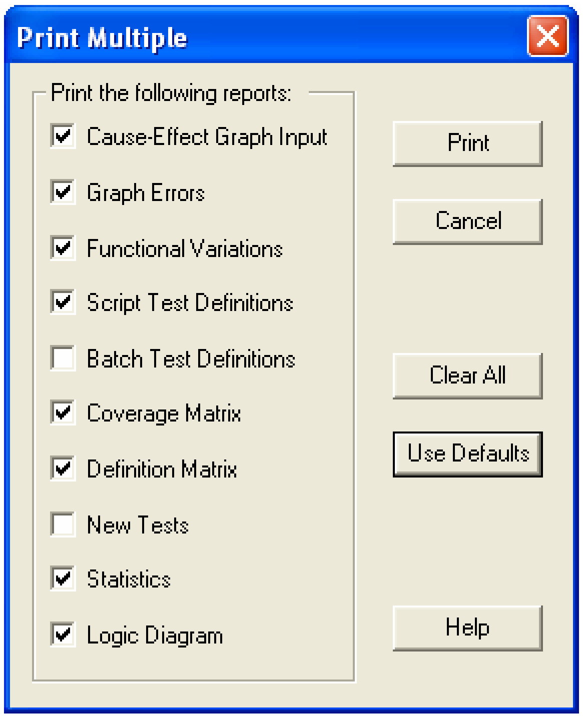
Coverage Matrix
Definition Matrix
Program Data

The following reports have no export option:

Cause-Effect Graph (the text API data)
Logic Diagram

## *5.17 Printing Multiple Reports*

For any given graph you can print out multiple reports at one time. Go to File → Print Multiple. The following dialog will appear:


Print Multiple

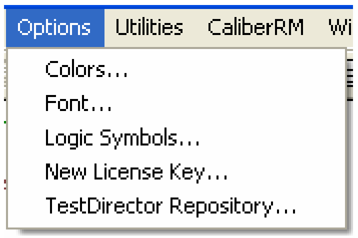
Print the following reports:
- [✓] Cause-Effect Graph Input
- [✓] Graph Errors
- [✓] Functional Variations
- [✓] Script Test Definitions
- [ ] Batch Test Definitions
- [✓] Coverage Matrix
- [✓] Definition Matrix
- [ ] New Tests
- [✓] Statistics
- [✓] Logic Diagram

**Print Multiple Dialog**

You can select any set of reports and the entire set will print back to back. The set shown in the example is the default set.

# 6. Options

When you select Options you get the following dialog:



**Options Dialog**

**Colors** allows you to control what colors to use to highlight various things in the RBT reports.

**Font** allows you to select what font to use for the various reports.

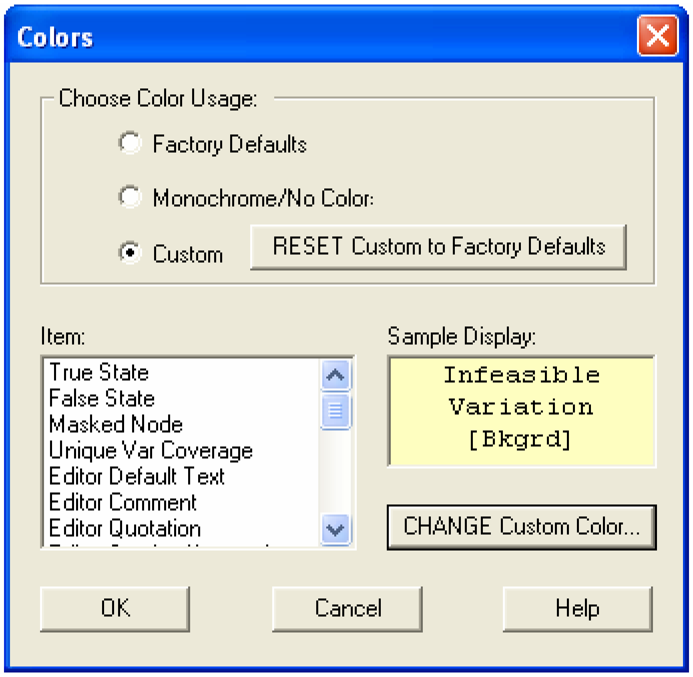
**Logic Symbols** allows you to select between the three symbols sets on the Logic Diagram (note that this does not affect the choice made for the graph in RBTg).

**New License Key** allows you to enter a new key for RBT.

**Test Director Repository** allows you to specify the location of TD.

## *6.1 Colors*

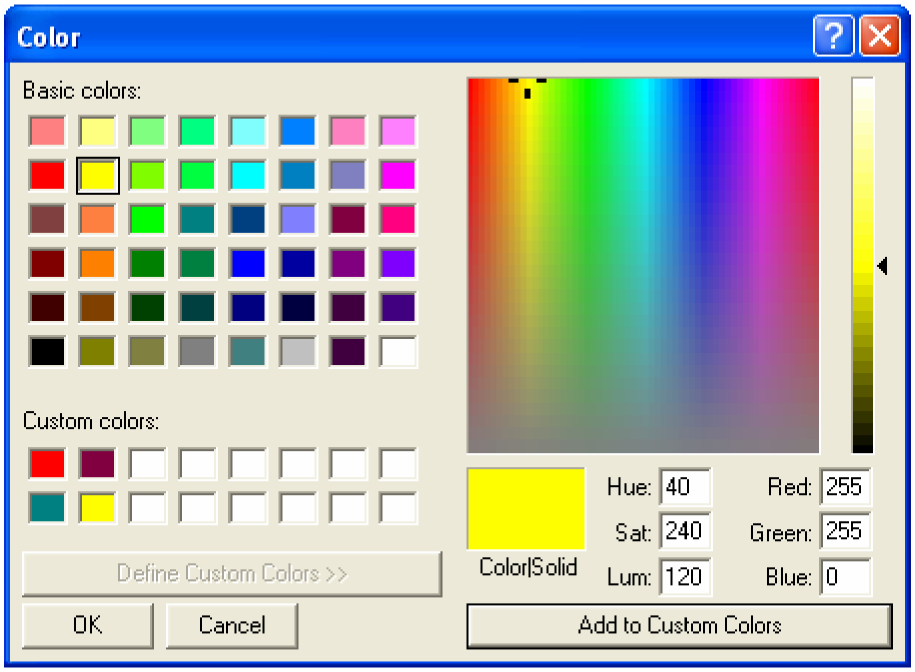
Selecting Options → Colors gives you the following dialog:



Choose Color Usage: Factory Defaults [✓ Custom] Monochrome/No Color:

**Colors Dialog**

With this you can tailor the look and feel of the various RBT reports. For example, on the Test Definition Matrix you can decide what colors to use for the True, False, and Masked states. For the Functional Variations report and Coverage Matrix you can specify what color to use as a background for infeasible, untestable, and untested variations.



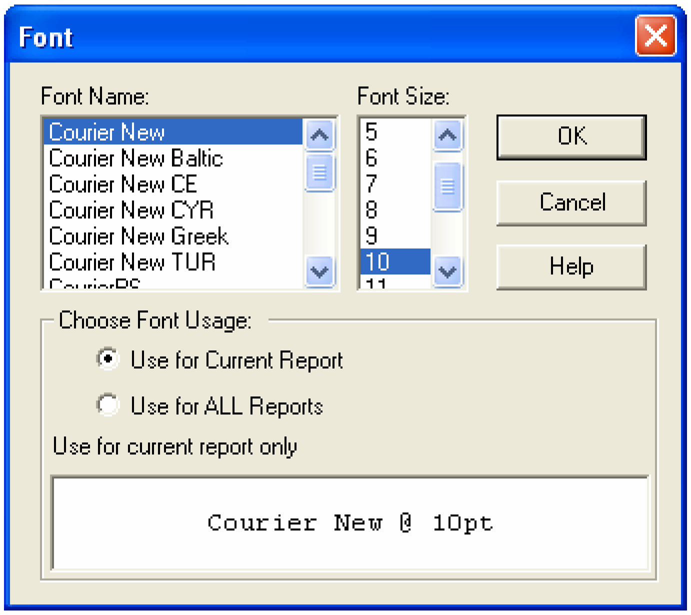
Hue: 40 Sat: 240 Lum: 120 Red: 255 Green: 255 Blue: 0

**Change Custom Colors**

Using the CHANGE Customer Color button will display a full color spectrum, allowing you to create any color you want – have fun.

## 6.2 Font

Selecting Options → Font gives you a classic font dialog:



[✓ Use for Current Report]

Use for ALL Reports

Use for current report only

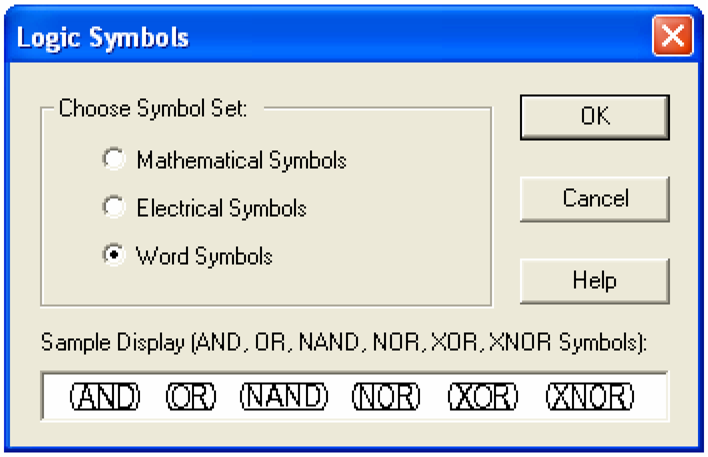
Courier New @ 10pt

**Font Dialog**

You can specify a font to use for all of the reports or just the one that is currently displayed.

## 6.3 Logic Symbols

As in RBTg, you can choose between the three symbol sets for the Logic Diagram report. Selecting a symbol set only effects the Logic Diagram, not the graph drawn in RBTg. This feature uses the RBT.ttf font that is installed along with RBT. Sometimes the font does not display properly when you first try this option. You then need to go into My Computer, select Fonts, scroll to the RBT.ttf font, and double click on it – i.e. this lets Microsoft OS know that the font exists. The font will then display. After that the font will display properly in the Logic Diagram.



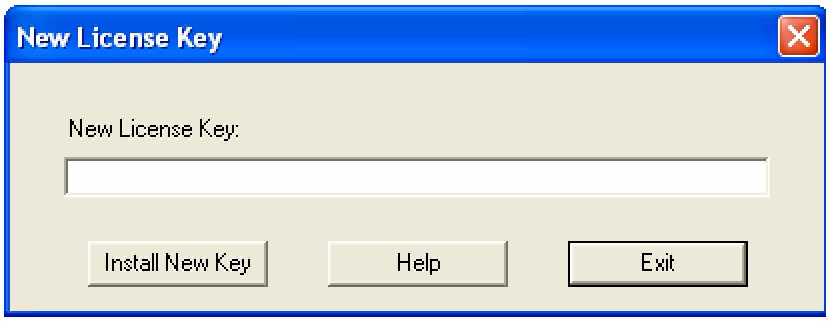
Choose Symbol Set:
Mathematical Symbols
Electrical Symbols
[✓ Word Symbols]

Sample Display (AND, OR, NAND, NOR, XOR, XNOR Symbols):
(AND) (OR) (NAND) (NOR) (XOR) (XNOR)

**Logic Symbols**

## 6.4 New License Key

This option allows you to enter a new key for RBT. This would mainly be used in the case where the user had an evaluation key that was being extended or converted to a permanent key.

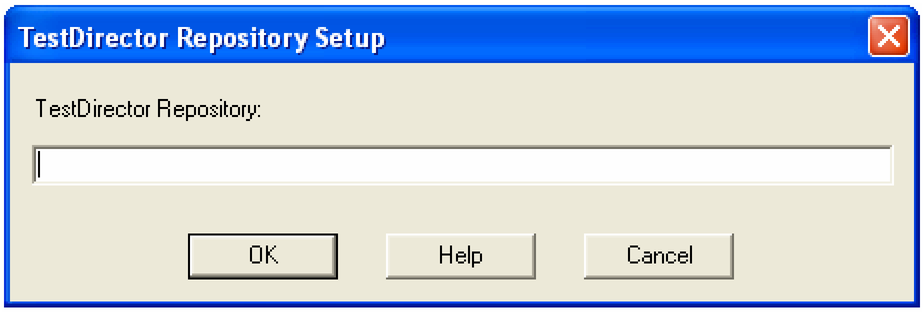


New License Key:

**New License Key**

## *6.5 TestDirector Repository*

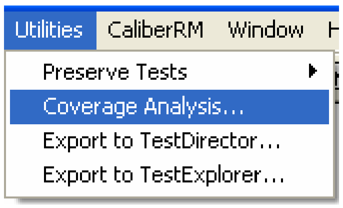
RBT can export the tests it has designed to the Test Planning section of Mercury Interactive's TestDirector tool. In this option you define the full path name where RBT can find TestDirector. The export to TestDirector will be discussed in the Utilities Chapter.



**Test Directory Setup**

# 7. Utilities

RBT has a number of useful utilities built in. Selecting the Utilities pull down gives you:



**Utilities**

**Preserve Tests** accesses the options for maintaining the test cases designed by RBT. This has already been discussed in the chapter on managing tests.

**Coverage Analysis** allows you to evaluate your test status based on what tests have passed or failed. It also allows you to optimize your overall test planning.

**Export to TestDirector** exports the RBT designed tests to Mercury Interactive's test management tool.

**Export to TestExplorer** exports the RBT designed tests to Sirius-SQA's test management tool.

## *7.1 Preserve Tests*

See Chapter 4 – Creating and Managing Test Cases.

## *7.2 Coverage Analysis*

The Coverage Analysis utilities can do two things. First, they can calculate what the coverage is for any subset of the tests. Second, they can determine what the optimal coverage is for any subset of the tests. There are two coverage measurements used by RBT: Weak Coverage and Strong Coverage.

Weak Coverage – If a functional variation was included in one or more test cases that were successful, then it is considered covered under Weak Coverage.

Strong coverage – If all of the functional variations for a given operator were in one or more successful tests then they are all considered covered under Strong Coverage.

However, if one or more of the variations from a given operator was not in any successful test, then none of the variations for that operator are considered covered by Strong Coverage.

Let us use an example to clarify this. Figure 1 shows a simple application rule that states that if you have A or B or C you should produce D. The test variations to test are shown in Figure 2. The "dash" just means that the variable is false. For example, the first variation is A true, B false, and C false, which should result in D true.

Let us assume that there are two defects in the code that implements our A or B or C gives us D rule. No matter what data you give it, it thinks A is always false and B is always true. There is no Geneva Convention for software that limits us to one defect per function.

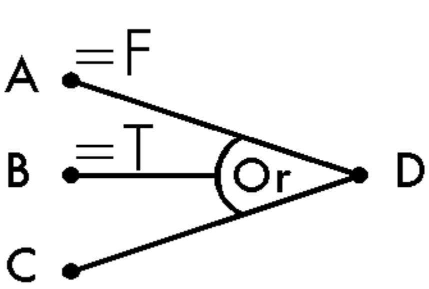
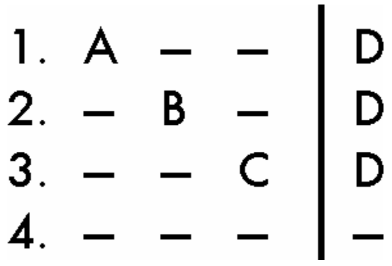


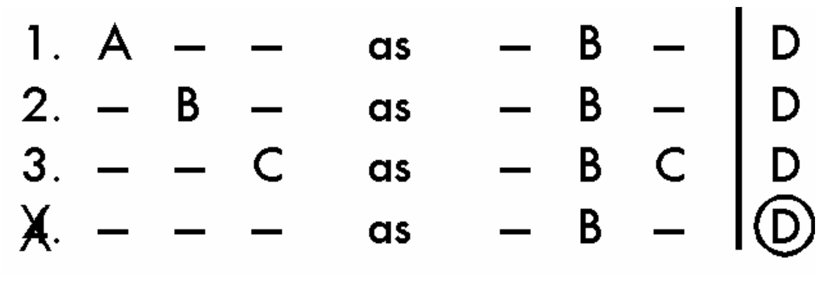
**Figure 1 - Simple "OR" Function With Two Defects**



| | | | | |
|---|---|---|---|---|
| 1. | A | – | – | D |
| 2. | – | B | – | D |
| 3. | – | – | C | D |
| 4. | – | – | – | – |

**Figure 2 - Required Functional Variations For The "OR" Operator**

Figure 3 shows the results of running the tests. When we run test variation 1 the software says A is not true, it is false. However, it also says B is not false, it is true. The result is we get the right answer for the wrong reason. When we run the second test variation we enter B true, which the software always thinks is the case – we get the right answer. When we enter the third variation with just C true, the software thinks both B and C are true. Since this is an inclusive "or," we still get the right answer. We are now reporting to management that we are three quarters done with our testing and everything is looking great. Only one more test to run and we are ready for production. However, when we enter the fourth test with all inputs false and still get D true, then we know we have a problem.

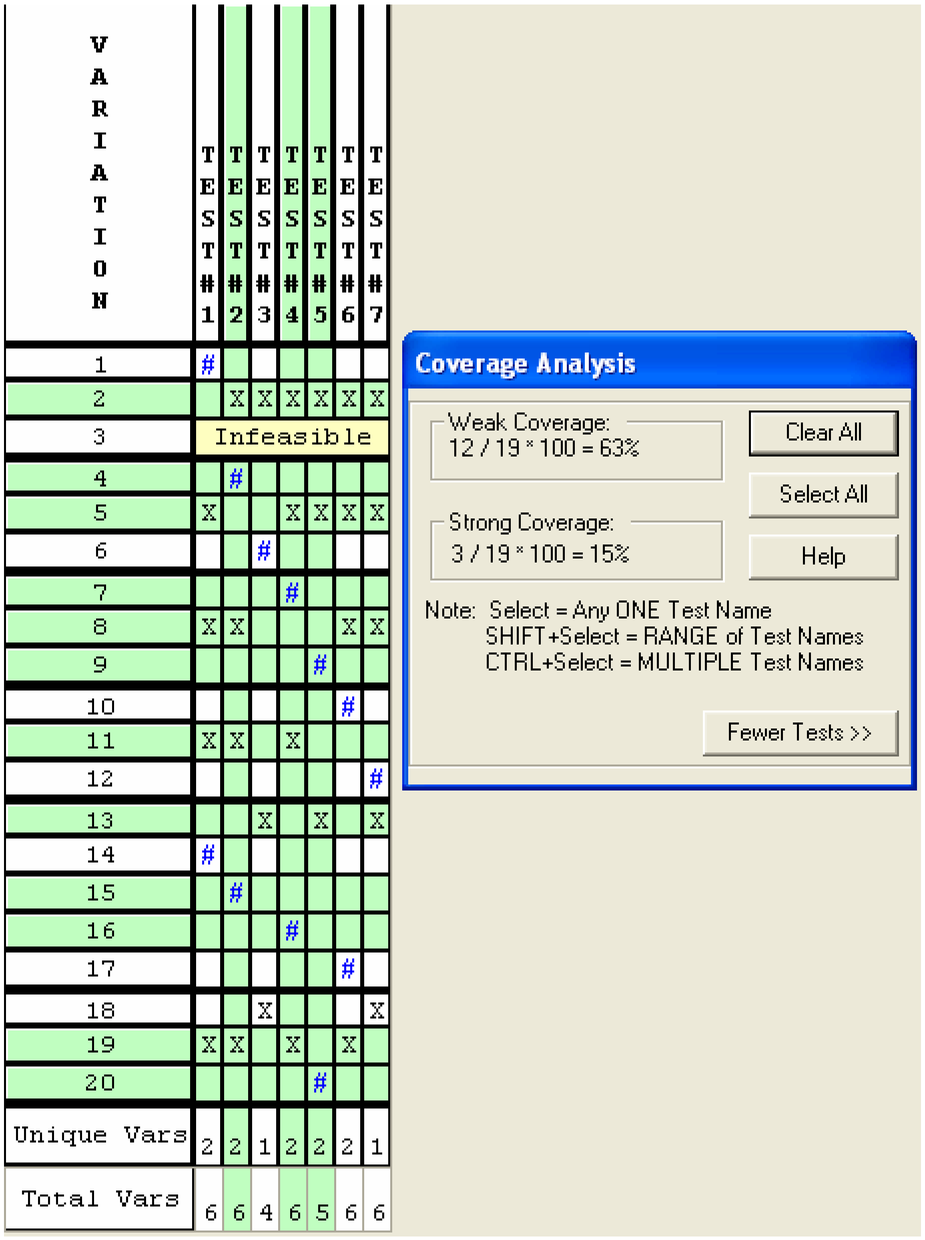


| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1. | A | — | — | as | — | B | — | D |
| 2. | — | B | — | as | — | B | — | D |
| 3. | — | — | C | as | — | B | C | D |
| ~~4~~. | — | — | — | as | — | B | — | D |

**Figure 3 - Variable "B" Stuck True Defect Found By Test Case 4**

Using this example, after we had run the four tests we would have had a Weak Coverage of 75%. Three out of the four variations were in tests that appeared to be successful. However, we would have had a Strong Coverage of 0% since all four variations were not in a successful test. Strong Coverage is what should be reported to Management to make Go – No Go decisions about releasing software.

Selecting the Coverage Analysis Utility will bring up the Coverage Matrix and a dialog showing the calculated coverage.



| VARIATION | TEST #1 | TEST #2 | TEST #3 | TEST #4 | TEST #5 | TEST #6 | TEST #7 |
|---|---|---|---|---|---|---|---|
| 1 | # | | | | | | |
| 2 | | X | X | X | X | X | X |
| 3 | Infeasible | | | | | | |
| 4 | | # | | | | | |
| 5 | X | | | X | X | X | X |
| 6 | | | # | | | | |
| 7 | | | | # | | | |
| 8 | X | X | | | | X | X |
| 9 | | | | | # | | |
| 10 | | | | | | # | |
| 11 | X | X | | X | | | |
| 12 | | | | | | | # |
| 13 | | | X | | X | | X |
| 14 | # | | | | | | |
| 15 | | # | | | | | |
| 16 | | | | # | | | |
| 17 | | | | | | # | |
| 18 | | | X | | | | X |
| 19 | X | X | | X | | X | |
| 20 | | | | | # | | |
| Unique Vars | 2 | 2 | 1 | 2 | 2 | 2 | 1 |
| Total Vars | 6 | 6 | 4 | 6 | 5 | 6 | 6 |

Coverage Analysis

Weak Coverage:
12 / 19 * 100 = 63%

Strong Coverage:
3 / 19 * 100 = 15%

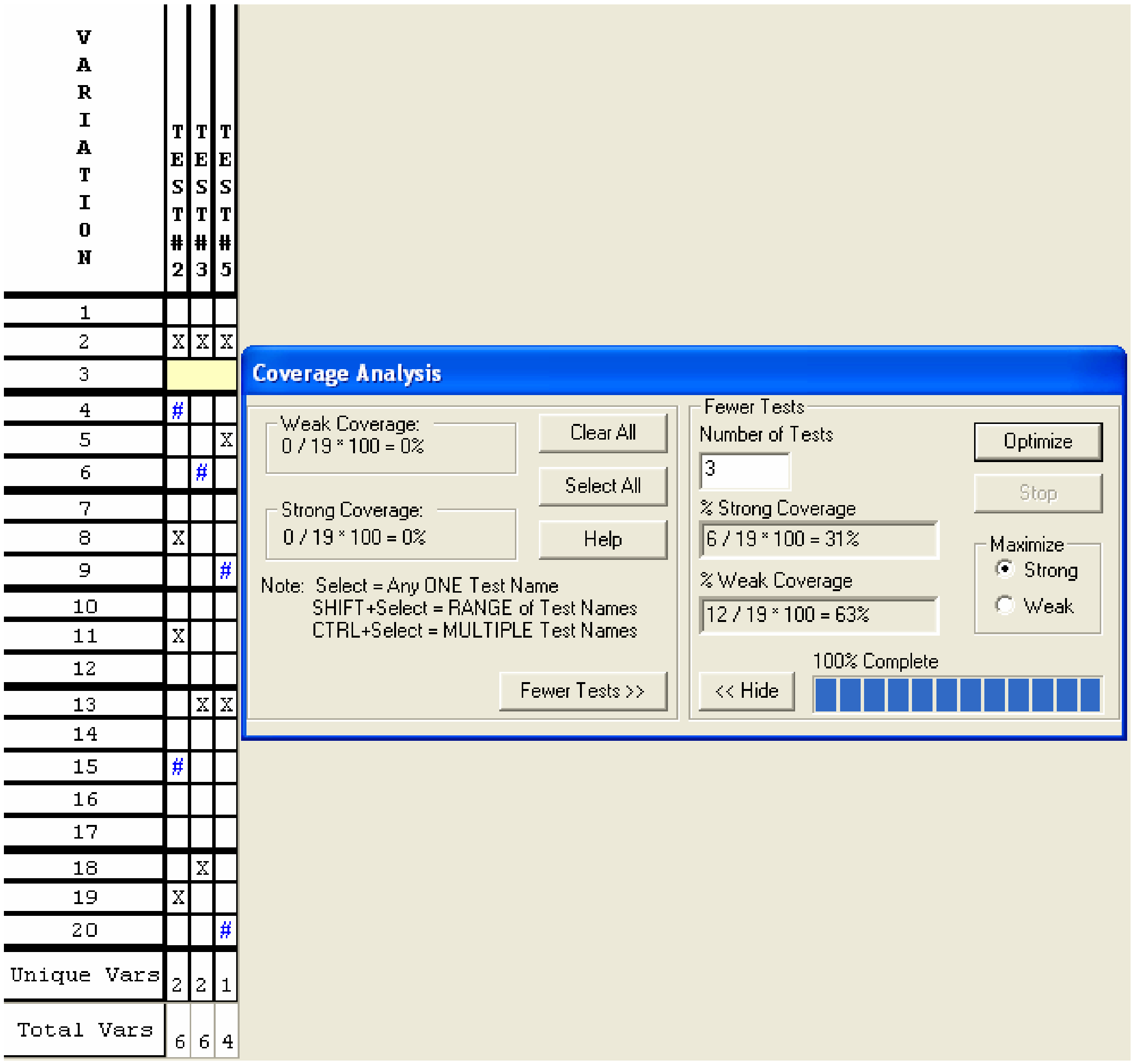
Note: Select = Any ONE Test Name
SHIFT+Select = RANGE of Test Names
CTRL+Select = MULTIPLE Test Names

**Coverage Analysis Dialog**

In this example (using Torder4) three tests have been marked as being successful – the ones in green. To mark a test has having been successful just click on that column. The dialog follows MS Windows conventions. If you select a column, it highlights. If you select another column, only the new column is highlighted. To select more than one test keep the CTRL button pressed as you select additional ones. To select a range of columns keep the SHIFT button pressed while you select the first and last column in the range.

With these three of the seven tests the current status is Weak Coverage = 63% and Strong Coverage = just 15%.

Notice on the dialog the “Fewer Test >>” option. Selecting this brings up a supplemental dialog:



| VARIATION | TEST #2 | TEST #3 | TEST #5 |
|---|---|---|---|
| 1 | | | |
| 2 | X | X | X |
| 3 | | | |
| 4 | # | | |
| 5 | | | X |
| 6 | | # | |
| 7 | | | |
| 8 | X | | |
| 9 | | | # |
| 10 | | | |
| 11 | X | | |
| 12 | | | |
| 13 | | X | X |
| 14 | | | |
| 15 | # | | |
| 16 | | | |
| 17 | | | |
| 18 | | X | |
| 19 | X | | |
| 20 | | | # |
| Unique Vars | 2 | 2 | 1 |
| Total Vars | 6 | 6 | 4 |

Coverage Analysis

Weak Coverage: 0 / 19 * 100 = 0%

Strong Coverage: 0 / 19 * 100 = 0%

Note: Select = Any ONE Test Name
SHIFT+Select = RANGE of Test Names
CTRL+Select = MULTIPLE Test Names

Fewer Tests

Number of Tests: 3

% Strong Coverage: 6 / 19 * 100 = 31%

% Weak Coverage: 12 / 19 * 100 = 63%

Maximize: [✓ Strong] Weak

100% Complete

**Fewer Tests Dialog**

This feature allows you to enter in a number less than or equal to the number of total tests and have RBT determine which is the optimal subset of tests – i.e. which tests would give you the greatest possible coverage.

The overall coverage feature is primarily used to measure and report test status. As a tester I love being able to tell Management, quantitatively, the status of testing. We take these numbers for each function and put them on spreadsheets. We can then calculate the overall coverage for the system. If Management wants to deploy the application prematurely, we ask them to sign these spreadsheets so we have a record of the status at the time of deployment. If we said something was tested and defects are found in production in that area, then that is our problem. However, if we made it clear that something was not yet finished testing and defects are later found, then that is Management's problem.

The Fewer Tests feature is used for two purposes. First, if Management is not giving you enough time to build and run all of the necessary tests, you can use this feature to select the best set of tests possible within the constraints you have been given. In the above example, if there is only time for creating three of the seven tests then we should choose tests 2, 3, and 5. We can also take this information to Management and explain ahead of time what the best we will be able to do will be. If 31% Strong Coverage is the best we can do, maybe this is not a good decision to limit the time we need.

The second use is to optimize the testing effort even where we do have time to create and run all of the needed tests. You would use this feature to decide which tests to build and run first. That will give you the greatest coverage in the shortest amount of time.

You can save this set of tests using Utilities → Preserve Tests → Save As.

## *7.3 Export To TestDirector*

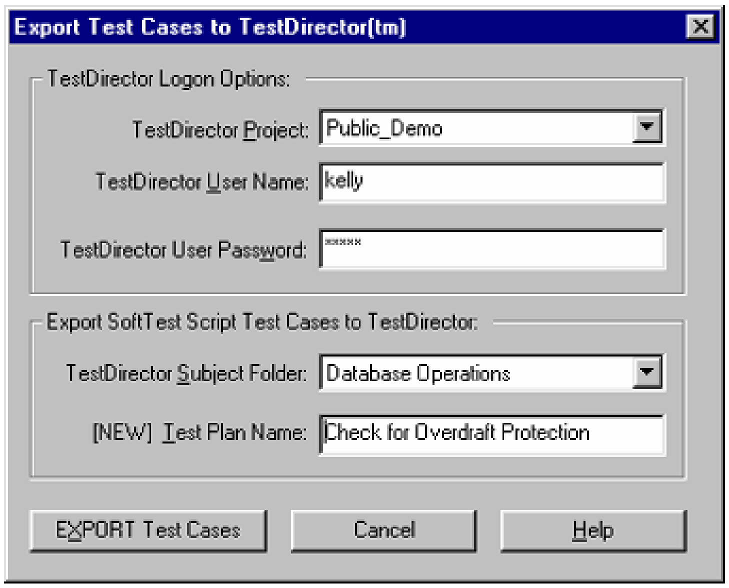
Mercury Interactive Corporation provides a suite of software applications that support automated software testing and project management. Their TestDirector product helps you organize and manage all phases of the software testing process, including planning tests, executing tests and tracking defects. TestDirector also provides facilities for developing the control scripts required to facilitate the automated execution of your software using their WinRunner product line. Additional third-party products are available which interface with the TestDirector test management tool. Further details regarding Mercury Interactive's offerings may be obtained from their website at http://www.merc-int.com.

TestDirector maintains a project management database of tests that cover all aspects of your application's functionality. To meet the various goals of a project, you organize the tests from your project database into unique groups. One of the basic components of TestDirector's management of the testing process is the test plan. A test plan identifies the objectives to be accomplished when testing your software. A test plan may be broken up into multiple steps, with each step containing a description of the step and its expected results. After a test plan has been developed, a TestDirector Generate Test Script facility may be utilized to generate a skeleton script which is used to control the test execution of the software. The script test case definitions generated by BenderRBT may be directly exported to a TestDirector test plan. The test plan then may be converted to a skeleton test execution script. On further instrumentation of the script with the code necessary to execute and validate the tests based on the cause and effect conditions specified, a suite of tests will be available to you containing the minimum number of tests necessary to validate the defined functionality of your system.

If the TestDirector facilities are accessible to you from your system, BenderRBT is able to locate and communicate with the TestDirector database. When errors are encountered communicating with the TestDirector database, one or more diagnostic messages are presented to help you resolve the communication failure.

The Export to TestDirector command on the Utilities menu is not enabled unless there are script test case definitions available for the active cause-effect graph file (the test case generation function must have been successfully completed).

When you select the Export to TestDirector command, the dialog box on the following page is displayed.


Export Test Cases to TestDirector(tm)
TestDirector Logon Options:
Export SoftTest Script Test Cases to TestDirector:
EXPORT Test Cases
Cancel
Help

TestDirector Project: Public_Demo
TestDirector User Name: kelly
TestDirector User Password: *****
TestDirector Subject Folder: Database Operations
[NEW] Test Plan Name: Check for Overdraft Protection

**Export to TestDirector**

On first use of this function, the edit boxes are empty. On subsequent usage, each of the boxes except the User Password and Test Plan Name will contain the values specified for the previous successful export of test cases.

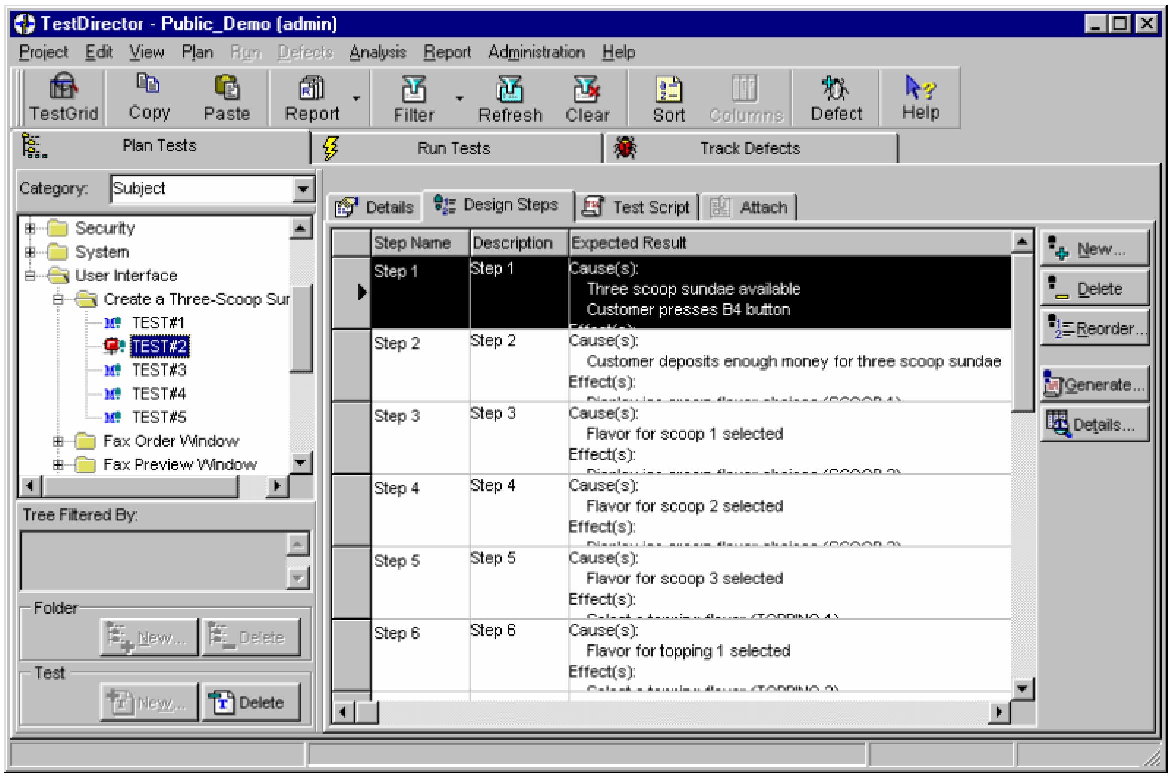
Begin by establishing the TestDirector logon options. A list of TestDirector Projects is available in the drop-down list. Ensure that the desired project name is selected. Enter your TestDirector user name and password to authorize your access to the TestDirector facilities.

A drop-down list containing the names of the folders in the TestDirector Project (specified above) is maintained in the box labeled TestDirector Subject Folder. Select the folder in which you would like the exported test cases to reside. Finally, specify a folder name that you would like to assign to the new test plan that you are about to create. Note that this folder name must not match any previously existing test plan name in the Subject Folder of the Project specified. (In other words, there is no test plan facility provided by BenderRBT, only an add facility. Facilities provided by TestDirector may be used to accomplish the deletion, renaming, copying and moving of test plans.)

When entries have been made in the minimum required fields in the dialog, then the Export Test Cases button will be activated. When you select this button, a connection to the TestDirector database is made and the BenderRBT Script test case definitions are exported to the named test plan folder. This process requires ten to fifteen seconds of processing time (or more, depending upon the speed of the processor, the size and number of the script test case definitions, and the overall size of the TestDirector database). If any problems are encountered during this process, a diagnostic message containing a statement of the problem is presented.

**Warning:** When you export your tests to TestDirector a folder will be created. Your TD privileges must be set up in such a way that you are allowed to create folders. Otherwise the export will fail. When it does fail for this reason TestDirector does not give back any specific return code that allows us to assist you in debugging the problem.

The following is an image of the TestDirector application displaying one of the test plans from the Uncle Mike's Ice Cream Machine example (TOrder4.rbt), which has been exported from BenderRBT.


TestDirector - Public_Demo (admin)
Project Edit View Plan Run Defects Analysis Report Administration Help
TestGrid Copy Paste Report Filter Refresh Clear Sort Columns Defect Help
Plan Tests Run Tests Track Defects
Category: Subject
Security
System
User Interface
Create a Three-Scoop Sur
TEST#1
TEST#2
TEST#3
TEST#4
TEST#5
Fax Order Window
Fax Preview Window
Tree Filtered By:
Folder
New... Delete
Test
New... Delete
Details Design Steps Test Script Attach
New... Delete Reorder... Generate... Details...

| Step Name | Description | Expected Result |
|---|---|---|
| Step 1 | Step 1 | Cause(s):<br>Three scoop sundae available<br>Customer presses B4 button |
| Step 2 | Step 2 | Cause(s):<br>Customer deposits enough money for three scoop sundae<br>Effect(s): |
| Step 3 | Step 3 | Cause(s):<br>Flavor for scoop 1 selected<br>Effect(s): |
| Step 4 | Step 4 | Cause(s):<br>Flavor for scoop 2 selected<br>Effect(s): |
| Step 5 | Step 5 | Cause(s):<br>Flavor for scoop 3 selected<br>Effect(s): |
| Step 6 | Step 6 | Cause(s):<br>Flavor for topping 1 selected<br>Effect(s): |

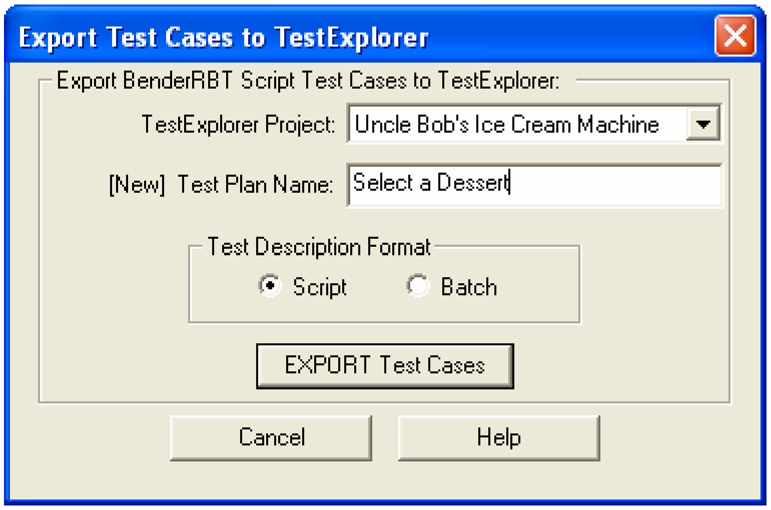
**Tests Exported to TestDirector**

## *7.4 Export to TestExplorer*

This feature works the same way as the export to TestDirector. TestExplorer is a test management tool aimed at manual testing. More information can be found about the product and the company at http://www.Sirius-SQA.com.

Unlike TestDirector, you do not need to tell RBT where TestExplorer is located. Each desktop installation of TestExplorer has a configuration file in the system directory that contains the path to the root drive where all projects reside. By default, the database is on the local drive, but it can reside on any network shared drive - in team environments it likely would. However, the configuration file lives on the local desktop, so each desktop knows where its database is. Consequently, different teams can have different databases as well. The dll is registered on the computer where RBT is located and finds this configuration file, which tells RBT where the actual database is. This configuration file is created when TestExplorer is installed. The implication is that for the integration to work, TestExplorer has to be installed and properly configured on the desktop. The dll does in fact validate this. So Tester-1, with TestExplorer on his desktop and RBT, can just ask Tester-2, who only has RBT on his desktop, to dump test cases into TestExplorer. Tester-2's RBT would simply inform him that TestExplorer was not installed. RBT and Test Explorer are using COM to communicate. When Test Explorer is installed, the COM server is registered with Windows, and after that, Windows knows where to find it.

Selecting Utilities → Export to TestExplorer brings up the following dialog:


Export Test Cases to TestExplorer

Export BenderRBT Script Test Cases to TestExplorer:

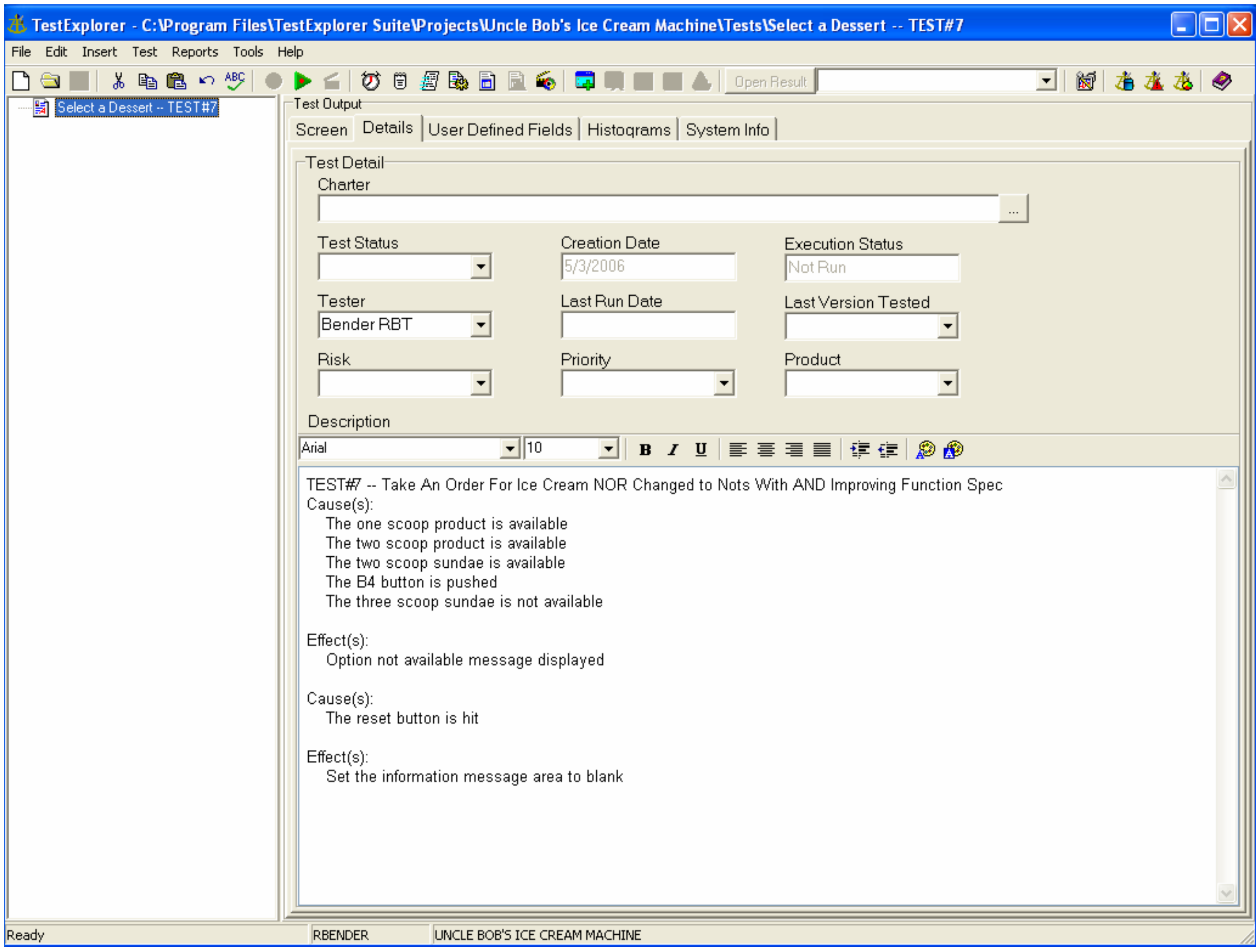
TestExplorer Project: Uncle Bob's Ice Cream Machine

[New] Test Plan Name: Select a Dessert

Test Description Format: [✓ Script] Batch

**Export to TestExplorer Dialog**

Once the tests have been exported, you can now find them in TestExplorer. Opening up the project and selecting one of the tests will show:


TestExplorer - C:\Program Files\TestExplorer Suite\Projects\Uncle Bob's Ice Cream Machine\Tests\Select a Dessert -- TEST#7
File Edit Insert Test Reports Tools Help
Open Result
Select a Dessert -- TEST#7
Test Output
Screen Details User Defined Fields Histograms System Info
Test Detail
Charter
Test Status
Creation Date
Execution Status
Tester
Last Run Date
Last Version Tested
Risk
Priority
Product
Description
Arial
10
Ready
RBENDER
UNCLE BOB'S ICE CREAM MACHINE

Creation Date: 5/3/2006
Execution Status: Not Run
Tester: Bender RBT

TEST#7 -- Take An Order For Ice Cream NOR Changed to Nots With AND Improving Function Spec
Cause(s):
The one scoop product is available
The two scoop product is available
The two scoop sundae is available
The B4 button is pushed
The three scoop sundae is not available

Effect(s):
Option not available message displayed

Cause(s):
The reset button is hit

Effect(s):
Set the information message area to blank

**Tests in TestExplorer**

# 8. CaliberRM Integration

Borland's CaliberRM is a collaborative, Internet-based requirements management system that facilitates more effective requirement definition and management throughout the development cycle. Providing a centralized requirement repository and automatic change notification, CaliberRM enables better collaboration and communication among project teams, assisting them in identifying and eliminating requirement errors earlier in the application lifecycle. CaliberRM also allows team members to compare project baselines to easily manage scope creep and identify other factors which may affect schedules and budgets. CaliberRM's support for reusable requirements ensures that project teams can build from previous experience and applications, enabling more rapid development and better use of resources. With CaliberRM, organizations can instill discipline in their development cycle through a structured requirements management process, minimizing the cost of reworks due to requirement errors, focusing team members on the project scope and decreasing the likelihood of project failures and overruns.

The basic CaliberRM environment centers on the use of CaliberRM and Framework Administrator servers running on a Windows 2000, 2003 or XP server or workstation. Users then may access the repository details from their local (or remotely connected) workstation. This environment must be established and working prior to attempting to link with it from a BenderRBT application.

The primary link between BenderRBT and CaliberRM is the fully-qualified file name of the cause-effect graph file. For every requirement managed by CaliberRM, references to one or more cause-effect graph files may be recorded on the references tab.

Note: Since the fully-qualified cause-effect graph file name includes the device identification (e.g., the R: portion in the fully qualified file name R:\icecream\Menu.ceg), it is necessary for the cause-effect graph files to reside on a shared network device which is accessible by all users using the same device identification. In more technical terms: the device must be mapped to the same identification letter.

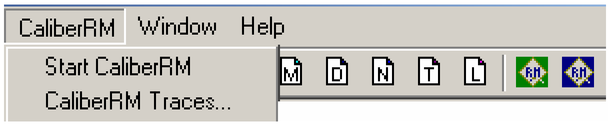
Within BenderRBT, users are able to:

• display a list of CaliberRM requirements that are linked to an open .ceg file.
• launch the CaliberRM Viewer.
• launch CaliberRM.
• drag a requirement from the CaliberRM requirement tree and drop it in an open .ceg file in BenderRBT to create a traceability link between them.

## Start CaliberRM

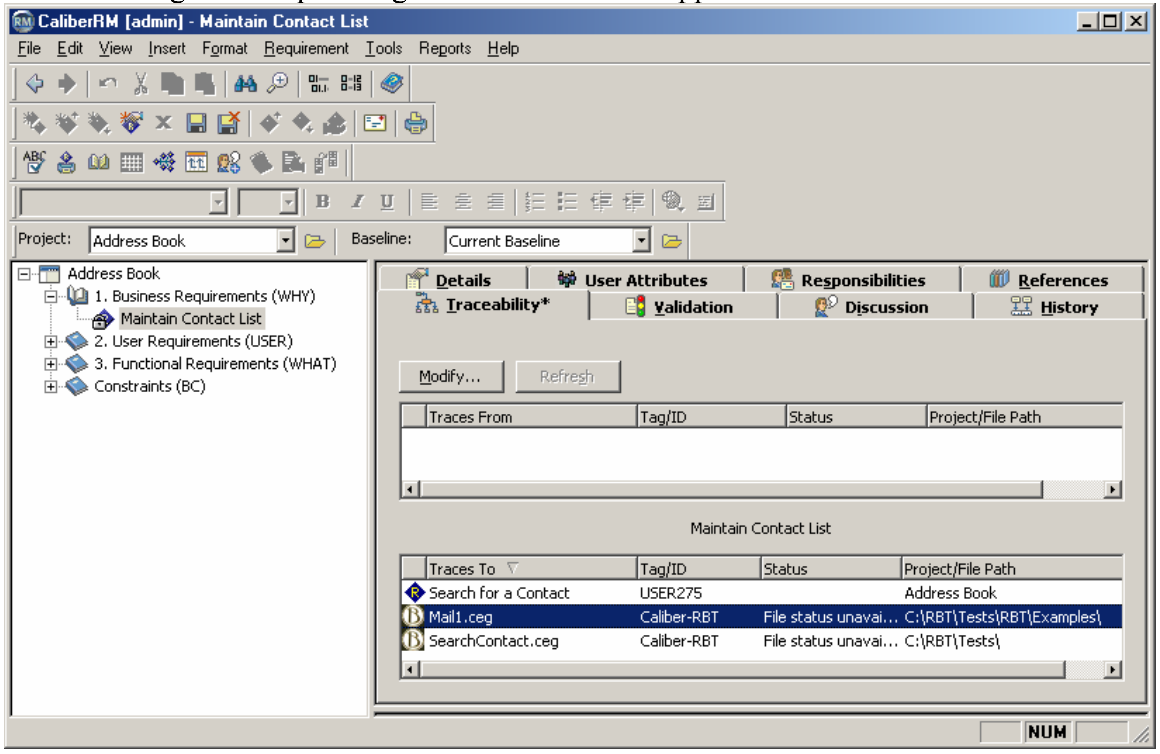
To launch CaliberRM from within BenderRBT:

1. Select CaliberRM > Start CaliberRM from the menu or click the (Blue) Start CaliberRM button on the toolbar.



2. The CaliberRM application is started.

The following is a sample image of the CaliberRM application.


CaliberRM [admin] - Maintain Contact List
File Edit View Insert Format Requirement Tools Reports Help
Project: Address Book
Baseline: Current Baseline
Address Book
1. Business Requirements (WHY)
Maintain Contact List
2. User Requirements (USER)
3. Functional Requirements (WHAT)
Constraints (BC)
Details
User Attributes
Responsibilities
References
Traceability*
Validation
Discussion
History
Modify...
Refresh
Maintain Contact List
NUM

| Traces From | Tag/ID | Status | Project/File Path |
|---|---|---|---|

| Traces To | Tag/ID | Status | Project/File Path |
|---|---|---|---|
| Search for a Contact | USER275 | | Address Book |
| Mail1.ceg | Caliber-RBT | File status unavai... | C:\RBT\Tests\RBT\Examples\ |
| SearchContact.ceg | Caliber-RBT | File status unavai... | C:\RBT\Tests\ |

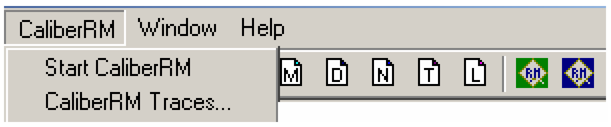
**CaliberRM**

**CaliberRM Traces Command**

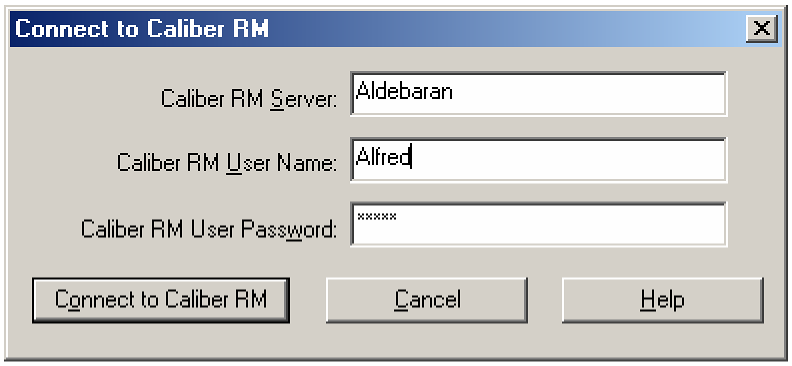
To display requirements that trace to an open .ceg file:

1. Open a .ceg file in BenderRBT.

2. Select CaliberRM > CaliberRM Traces from the menu or click the (Green) CaliberRM Traces button on the toolbar.



3. for security purposes, BenderRBT activates a CaliberRM Login Dialog, to verify your identity:

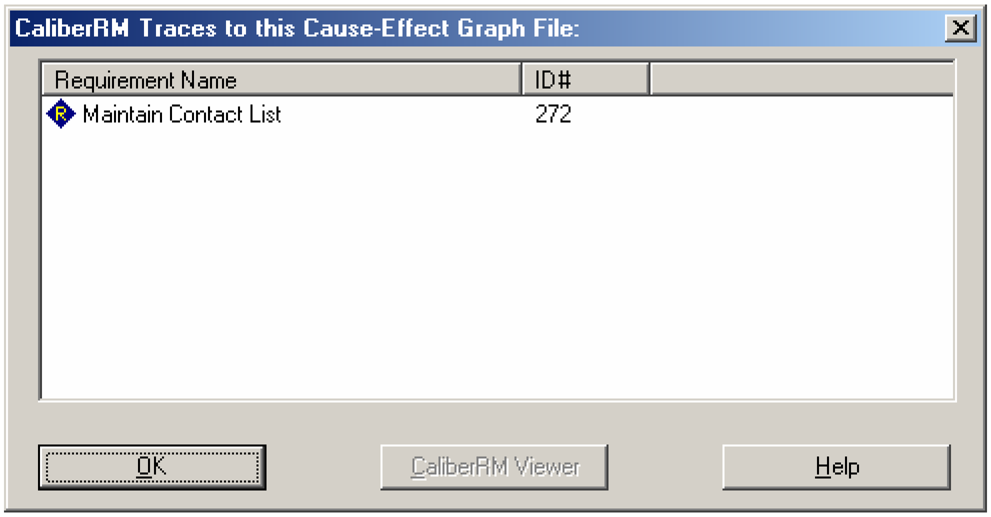


Caliber RM Server: Aldebaran
Caliber RM User Name: Alfred
Caliber RM User Password: *****

Enter the name of the server, your user name and your password. Click "Connect to Caliber RM".

4. BenderRBT initiates an inquiry to CaliberRM, asking for the return of any and all references to the currently open input cause-effect graph file.

Further BenderRBT processing is suspended while this transaction is processed by CaliberRM. Depending upon the current load on the network and the speed of the network server, this request may take a few seconds before a response is returned. If any communications errors are encountered, a separate dialog message is returned; otherwise, the CaliberRM Traces dialog box reports the identification number and title of each document in the CaliberRM repository that traces to the current cause-effect graph file.


CaliberRM Traces to this Cause-Effect Graph File:

| Requirement Name | ID# |
|---|---|
| Maintain Contact List | 272 |

**CaliberRM Traces**

5. Click OK to close the dialog box or click the CaliberRM Viewer to view details about the requirement. CaliberRM Viewer launches and displays the Requirement Viewer dialog box.

# 9. System Limits

The following are the system limits imposed by the present software design of BenderRBT:

• The maximum number of explicitly defined nodes is 4,090. A defined node is tallied for each cause or effect name referenced in a Relations statement. A node name appearing more than once in a Relations statement, or in more than one Relations statement, is only counted once. This count is also incremented once for each BenderRBT-generated intermediate effect name (*INT-xx**).
• The maximum number of causes in one connective is 64. A connective may be thought of as the ':-' symbol in the Relations statement (i.e., the connective between any effect node and its causes).
• The maximum number of intermediate effects (explicit plus implicit) in one relation is 64. (This is different from the above restriction: one intermediate effect node may lead to a maximum of 64 causes.)
• The maximum number of constrained nodes is 4,096. A constrained node is tallied for each occurrence of a node name appearing within all Constraint statements.
• The maximum number of functional variations is not limited.
• The maximum number of paths generated is not limited.
• The maximum number of test cases (old and new) is not limited.

The above limits apply to the composite graph—that is, the parent graph together with all of its subgraphs. If the limits are exceeded; or, if you wish to control the processing time required for any one graph, you may adapt the graph and the execution procedures on the following ways:
• The number of explicit nodes can be reduced by partitioning the specification and then processing subsets of the partitions separately.
• The number of causes linked to an effect can be reduced by dividing the connective into multiple simpler connectives.
• Any complex relation statements which exceed the effect limit can be divided into multiple, simpler statements.
• Each node specified in a constraint statement counts toward the limit, even if the same node is specified in another constraint statement. If this limit is exceeded, the graph must be partitioned and the partitions processed independently.
• The number of functional variations, paths and tests can be reduced by partitioning the specification and then processing subsets of the partitions separately.

There may be a limit imposed on the dynamic allocation of memory available for internal graph-dependent data. The specific limitation is dependent upon the operating system in use, coupled with consideration of the real versus virtual RAM available. If BenderRBT detects that the available memory space is insufficient for dynamic allocation, an error message is generated and execution terminates.

To circumvent this problem, you must take one or both of the following steps:

- Increase the system’s RAM size.
- Decrease the graph size through partitioning.

# 10. Diagnostic and Error Messages

## *10.1 Overview*

This section lists the diagnostic and error messages generated by BenderRBT, describes the conditions that may be responsible for each message and suggests corrective action.

All diagnostic messages are presented in the following format:

Ann—xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

where:

the **Ann** message identification prefix consists of one alphabetic character and two numeric digits; and the remaining portion is a short, free-form description of the diagnosis or error type.

The following alphabetic codes are used in the identification prefix and assigned for the message types indicated:

| Prefix CODE | Message TYPE |
|---|---|
| I | **I**nternal errors |
| L | **L**imit errors |
| S | statement **S**yntax (or **S**evere) errors |
| W | statement **W**arning messages |
| V | Infeasible **V**ariations |
| T | Infeasible and Untestable Variations, and other messages |

## *10.2 Internal Error Messages*

***Ixx—Internal BenderRBT engine failure***

It is possible, but unlikely, that an error message having an identification code starting with 'I' may be generated at any time during a **Design Tests** process. Good software coding standards call for an ELSE clause to be associated with every IF/THEN conditional statement. The occurrence of an Internal BenderRBT Engine Failure message indicates that there is a problem with the engine/run processor's internal logic.

If you encounter one of these messages, you should contact Bender & Associates as listed in **Appendix A**. Further run processing is terminated.

## *10.3 Limit Error Messages*

These messages indicate that some programmed system limit has been exceeded. See **Chapter 7: System Limits** for a definition of the established limits and the possible corrective actions when a limit is exceeded. Following any appropriate changes to the cause-effect graph file, a **Design Tests** function must be rerun.

***L20—Too many explicit nodes (at <nodename>)***
The number of explicit nodes exceeds the established limit. The last node name processed is shown in parentheses.

***L22—Too many causes in one connective***
The number of causes leading to an implicitly-defined (i.e., BenderRBT-generated *INT-xx**) or explicitly-defined effect node exceeds the established limit.

***L23—Too many effects in one relation***
The number of effects in one Relation exceeds the established limit.

## *10.4 Syntax (or Severe) Error Messages*

These messages alert you to syntax errors (also referred to as Severe errors) that BenderRBT cannot correct. Each faulty statement is printed with a message suggesting the apparent error. That statement is bypassed and the next statement's error messages processed. An error in one statement may cause an error in a succeeding statement. Because of the unprocessed statement(s), succeeding phases of the **Design Tests** process are not executable. You must correct the error(s) in the input cause-effect graph and re-execute the **Design Tests** request.

Note that if the cause-effect graph file was created using RBTg, then many of these errors are impossible to get. The properties sheets used to define the nodes edit the information as you enter it.

***S01—Tested cause is non-invokable: <nodename>***
Only the states of invokable causes can be specified in test statements.

***S02—Unable to open input cause-effect graph file: <fileSpec>***
The named cause-effect graph file cannot be opened for processing. You should check the disk drive, directory path and file name specified. Ensure that the file does, in fact, exist and that it is not currently being processed by another application. Restart the **Design Tests** process to re-attempt access to the file.

***S03—Circular logic involving node: <nodename>***
Cause-effect graphs can express only combinatorial relationships; no node can be related, directly or indirectly, to itself. Also, constraints cannot link causes to related effects.

***S04—Unexpected error during CLOSE of input: <fileSpec>***
During the termination of the phase that parses the cause-effect graph file, some abnormal error condition was encountered.
You should check the disk drive, directory path and file name specified. Ensure that the device and path do, in fact, exist and that the file name is not currently being processed by another application. Restart the **Design Tests** (or **Check Syntax** or **Revise Descriptions** process to re-attempt to parse the file.

***S05—Unable to open input cause-effect graph file: <fileSpec>***
The named cause-effect graph file cannot be opened for processing. You should check the disk drive, directory path and file name specified. Ensure that the file does, in fact, exist and that it is not currently being processed by another application.
Restart the **Design Tests** process to re-attempt access to the file.

***S06—Unexpected error during CLOSE of input: <fileSpec>***
During the termination of the phase which parses the cause-effect graph file, some abnormal error condition was encountered. You should check the disk drive, directory path and file name specified. Ensure that the device and path do, in fact, exist and that the file is not currently being processed by another application. Restart the **Design Tests** (or **Check Syntax** or **Revise Descriptions**) process to re-attempt to parse the file.

***S07—User requested 'Cancel' of run in progress***
You have specified that the current run in progress be canceled. No report outputs are available. Restart the **Design Tests** (or **Check Syntax** or **Revise Descriptions**) process to re-attempt to parse the file.

***S08—Unable to open input cause-effect graph file: <fileSpec>***
The named cause-effect graph file cannot be opened for processing. You should check the disk drive, directory path and file name specified. Ensure that the file does, in fact, exist and that it is not currently being processed by another application. Restart the **Design Tests** process to re-attempt to access the file.

***S09—Unexpected error during CLOSE of input: <fileSpec>***
During the termination of the phase that parses the cause-effect graph file, some abnormal error condition was encountered. You should check the disk drive, directory path and file name specified. Ensure that the device and path do, in fact, exist and that the file name is not currently being processed by another application. Restart the **Design Tests** (or **Check Syntax** or **Revise Descriptions**) process to re-attempt to parse the file.

***S10—Right comment symbol missing after: <line>***
A comment is opened but not closed within the statement line.

***S11—Left comment symbol missing near: <line>***
A comment is closed but not opened within the statement line.

***S12—Too many characters in statement following: <statement>***
No single statement can exceed 4,090 characters after multiple-space characters and tab characters are removed. The input statement following the statement given in the message appears (to BenderRBT) to exceed this limit. In all likelihood there is a missing apostrophe, or other delimiter, in the statement.

***S13—Quotation symbol missing after: <statement>***
A quotation is opened but not closed, or vice-versa, within a given input line.

***S14—Insufficient information in statement following: <statement>***

The information in the last statement before end-of-file is syntactically incomplete. This includes the following possible errors:

- a missing comment terminator '*/'
- a missing statement terminator '.'

***S15—Extraneous information in: <statement>***
Information other than a comment enclosed within /*.......*/ characters follows the terminating period in the statement.

***S16—Quotation too long in: <statement>***
The quoted string is too long. Shorten the quotation to no more than the maximum allowed, depending on whether a Title or Nodes description is involved. Holding descriptions to no more than 60 characters is recommended.

***S17—User requested 'Cancel' of run in progress***
You have specified that the current Run in progress be canceled.
No report outputs are available. Restart the **Design Tests** (or **Check Syntax** or **Revise Descriptions**) process to re-attempt to parse the file.

***S19—Expected header missing prior to: <statement>***
The statement is not a header, nor does it follow a header. Define the header.

***S20—Extraneous information in: <statement>***
A symbol having no significance in a title statement appears ahead of the period in the statement.

***S31—Node name used in Relations statement not defined or invalid: <nodename>***
The referenced node name has been defined using a nodes statement but not referenced by a Relations statement.

***S32—Node name too long in: <statement>***
The node name (defined in a Nodes statement) is too long.
Shorten the node name to the maximum allowed. The definition and use of node names of 12 characters or less is recommended.

***S34—Extraneous information in: <statement>***
A symbol having no significance in a Nodes definition statement appears ahead of the period in the statement.

***S35—Reserved word used for nodename: <nodename> in: <statement>***
A keyword reserved for BenderRBT's use has been used as a node name in a Nodes statement. Change or abbreviate the node name definition.

***S36—Null description defined for primary cause node: <nodename>***
Either a true or a false description must be defined for any primary cause node in a cause-effect graph.

***S37—Missing period in Nodes statement: <statement>***
A period is missing at the end of the Nodes statement indicated in the error message. A period is syntactically necessary to denote the termination of each node declaration statement.

***S39—Duplicate cause node reference in: <statement>***
The same cause node appears more than once in a relations statement. More specifically, the same cause node cannot be repeated in a relations statement where the nodes are connected by the same relational operator. Since this is syntactically and logically incorrect, the Relations statement must be corrected.

***S40—Cannot Relate the false-state of an effect name in: <statement>***
Only the true state of an effect node may be defined using a Relations statement. Remove the reserved keyword **not** from the beginning of the Relations statement and redefine the relationship as appropriate.

***S41—Previously-defined effect: <nodename> in: <statement>***
A given effect can be defined by only one Relations statement.

***S42—Logical operator missing in: <statement>***
A connective (or logical operator, such as AND, OR, NAND, NOR, XOR, XNOR) is missing from the Relations statement.

***S43—Conditional symbol missing in: <statement>***
*Explanation:* The second symbol in a relations statement is not a colon-dash (**:-** ).

***S44—Right parenthesis missing in: <statement>***
The right parenthesis expected at the end of a cause construct is missing in a relations statement.

***S45—Cause name missing/invalid in: <statement>***
A cause name referenced in a relations statement has not been defined using a nodes statement, or is otherwise invalid.

***S46—Effect name missing/invalid in: <statement>***
A valid, defined effect name is missing in the Relations statement.

***S47—Circular reference to <nodename> Effect within Relation: <statement>***
The effect name in the statement is repeated as a **not** cause name in the same Relations statement.

***S48—Use parentheses to clarify multiple operators in Relations statement for: <statement>***
Multiple or different logical operators (AND, OR, NAND, NOR, XOR, XNOR) have been used within the same Relations statement and no parentheses are specified to clarify the logical intent of the relationship. There is NO operator precedence supported within the cause-effect graphing language. In order to be totally unambiguous, parentheses must be inserted at the appropriate points in order to clarify the precedence of causes and their logical operators.

***S49—Superfluous parentheses used in Relations statement for: <nodename>***
One or more pairs of unnecessary parentheses have been encountered in a relations statement. Remove the superfluous parentheses. Parentheses are only required when there are multiple or different logical operators (AND, OR, NAND, NOR, XOR, XNOR) present in the same Relations statement.

***S50—Invalid constraint type in: <statement>***
The first symbol in the Constraints statement is not a valid constraint type (i.e., MASK, REQ, EXCL, INCL, ONE, ANCHOR).

***S51—Left parenthesis missing in: <statement>***
The second symbol in the Constraints statement is not a left parenthesis.

***S52—Node name missing/invalid/not Related: <nodename> in: <statement>***
An insufficient, undefined or reserved node name is specified in the Constraints statement; or, one or more node names in the constraint have not been referenced in a Relations statement.

***S53—Right parenthesis missing in: <statement>***
The statement is terminated without balancing the parentheses. Note that if this message occurs for a Constraints statement, then one or more nodes referenced in the Constraint also have not been referenced in a relations statement. Review the list of nodes presented in any W02 messages for nodes which are not used in the graph.

***S54—Comma missing in: <statement>***
A comma is required between node names in the Constraints statement.

***S55—Node name(s) repeated in: <statement>***
One or more node names have been repeated in the Constraints statement.

***S56—Extraneous information in: <statement>***
A symbol having no significance in a constraint appears ahead of
the period in the statement.

***S58—False object node state(s) not allowed in: <statement>***
The declaration of false node states for the objects of a Mask is neither required nor allowed. Only the subject node state may be declared in a Mask Constraint statement. Remove the **not** false-state declarations for each of the object nodes in the Constraints statement.

***S59—Subject of MASK also named as object in: <statement>***
The subject node name in a Mask constraint statement has also been named as an object (i.e., a node cannot mask itself).

***S60—Test name too long in: <statement>***
The test name is too long in the Tests statement. Shorten the test name to the maximum allowed. Test names of 12 characters or less are recommended.

***S61—Previously-defined test name in: <statement>***
The specified test name was also used to identify a preceding test in another Tests statement.

***S62—Equal sign missing or node not referenced in: <statement>***
The required delimiter (=) between the test name and the first cause state is missing in the Tests definition statement.

***S63—Invalid test name: <testname>***
A reserved name or symbol has been specified as the test name in a Tests definition statement.

***S64—Node specification duplicated in: <statement>***
The state of one or more causes is specified multiple times in the Tests definition statement.

***S66—Comma missing in: <statement>***
A comma is required as a delimiter between each pair of node names in the Tests statement.

***S70—Invalid file specification: <statement>***
The first symbol in the Subgraphs statement is a reserved name or symbol.

***S71—Non-existent SUBGRAPH file: <statement>***
The file specification in the Subgraphs statement points to a nonexistent file.

***S72—Extraneous information in: <statement>***
A symbol having no significance appears ahead of the period in the Subgraphs statement.

***S73—Inconsistent/duplicate subgraph pointer in: <statement>***
This reference in a Subgraphs statement is inconsistent with a preceding reference to the same subgraph; the two pointers convey different active or passive attributes to the subgraph.

***S74—Omit ".CEG" in Subgraphs statement: <statement>***
Inclusion of the .CEG file name suffix is neither required nor allowed (as the period in the file name will potentially confuse the cause-effect graph parser).

***S90—Case-sensitivity setting was changed***
A change has been made to the **Case-sensitivity** setting and cannot be accepted by the **Revise Descriptions** processor. Refer to the discussion of acceptable versus unacceptable changes in the **Revise Descriptions** section in Chapter 4. Change the **Case-sensitivity** setting back to its prior state (i.e., the state the setting was in when the input cause-effect graph file was originally processed). Then rerun the **Revise Descriptions** process until no Severe error messages are

generated; otherwise, it is necessary to run a full **Design Tests** process.

***S91—Unacceptable change involving node <nodename> encountered***
A change has been made to the definition of the named node that cannot be accepted by the **Revise Descriptions** processor. Refer to the discussion of acceptable versus unacceptable changes in the **Revise Descriptions** section in Chapter 4. Modify the cause-effect graph file back to its original contents, and/or make further corrections as deemed appropriate. Then rerun the **Run Descriptions** process until no Severe error messages are generated; otherwise, it is necessary to run a full **Design Tests** process.

***S92—Number of node names defined/referenced has changed***
An unacceptable change to the number of nodes statements present, or their subsequent reference via relations statements has been detected by the **Revise Descriptions** process. Refer to the discussion of acceptable versus unacceptable changes in the **Revise Descriptions** section in Chapter 4. Modify the cause-effect graph file back to its original contents, and/or make further corrections as deemed appropriate. Then rerun the **Revise Descriptions** process until no Severe error messages are generated; otherwise, it is necessary to run a full **Design Tests** process.

***S93—Unacceptable change involving <constraint type> (<nodename>,...) Constraint encountered***
Any change to a constraints statement cannot be accepted by the **Revise Descriptions** processor. Refer to the discussion of acceptable versus unacceptable changes in the **Revise Descriptions** section in Chapter 4. Modify the cause-effect graph file back to its original contents, and/or make further corrections as deemed appropriate. Then rerun the **Revise Descriptions** process until no Severe error messages are generated; otherwise, it is necessary to run a full **Design Tests** process.

***S94—Number of Constraints defined has changed***
Any change to a constraints statement cannot be accepted by the **Revise Descriptions** process. Refer to the discussion of acceptable versus unacceptable changes in the **Revise Descriptions** section in Chapter 4. Modify the cause-effect graph file back to its original contents, and/or make further

corrections as deemed appropriate. Then rerun the **Revise Descriptions** process until no Severe error messages are generated; otherwise, it is necessary to run a full **Design Tests** process.

## 10.5 Warning Messages

These messages identify suspicious, though not critical, observations about the graph. None of these warnings block execution of the succeeding run phases.

***W02—Defined node is not used in graph: <nodename>***
The specified node is not referenced by a Relations statement. It would be prudent, though not essential, to delete the Nodes definition statement which defines this node. When a data dictionary of node definitions is being used, this message is informational only.

***W03—Defined node is redefined in graph: <nodename>***
The specified node has been defined via nodes statements more than once in the cause-effect graph file stream. Ensure that the *last* definition for the node name listed in the message is the desired definition. When a data dictionary of node definitions is being used, the node listed confirms that this node has been overridden.

***W04—Can intermediate node be observed <nodename>?***
The specified node is an intermediate cause from which fanout paths later reconverge at an effect. It would be prudent, though not essential, to directly observe the states of this node in order to minimize the likelihood of untestable variations. Observability of intermediate nodes can be specified using the Nodes definition statement by adding either the OBS or FOBS keyword to the node definition statement for this node. If this is done, the **Design Tests** process must be re-executed.

***W19—TITLE statement Description field missing or misplaced***
The quoted description field is either not included in the title statement, or it is misplaced (i.e., there are other characters between the title keyword and the description field). It is not necessary that the input graph contain a Title description. If the description field is misplaced, then the description is not recorded.

***W20—Extraneous/Invalid/Obsolete Switch Info in TITLE statement***

Prior to Release 5.0 of BenderRBT, various run-time and report formatting options were passed to the program using “switch” keywords in the title statement. A switch keyword would begin with the forward slash character (/). For cosmetic purposes, you may wish to remove the switch keywords if processing of this cause-effect graph file is to take place. For purposes of upwards-compatibility, switch keywords in the title statement are ignored by Release 5.0.

***W21—False-state suppressed on OBServable Node: <nodename>***

The ‘/B’ bypass false-state condition has been specified using the nodes definition statement for the indicated nodename. This node name has also been flagged as an observable effect (using the OBS or FOBS keyword in the nodes definition statement). This is a warning message only. Be aware that even though this effect has been designated as *observable,* any time that the false-state effect is applicable to a test, it does not appear in either the script or batch test case definition listings. This situation would be plausible, for example, when the true-state of an effect is to display an error message on a terminal (i.e., it is neither necessary nor desirable that the absence or false-state of the error message not appearing be indicated in every test case definition). If it is important to the testing scenario that false-state conditions for this node name be observable and present in each test case definition, then remove the ‘/B’ designation and consider specifying a false-state description.

***W22—False-state suppressed on unconstrained node: <nodename>***

The ‘/B’ bypass false-state condition has been specified using the nodes definition statement for the indicated node name, which subsequently has not been included in a Requires, One-and-only-one, or Includes Constraint statement. This is a warning message only. Be aware that when the false-state of the node name is applicable to a test, it does not appear in either the batch or script test case definition listings. Unless this node is constrained with one or more other nodes, its definition may be logically inconsistent with the intended requirements specification. Review the requirements of the specified node name as they may be influenced by one or more other node names and, as appropriate, establish the logical Constraint requirements necessary.

***W23—REQ(<nodename>) subject node not a Primary Cause***
The subject node in a Requires Constraint statement should logically name a primary cause node (i.e., a node that is not also an effect node). To BenderRBT, this appears to be illogical and subsequent results are unpredictable. (Note: this condition may be designated as a Severe condition in a future release of BenderRBT, requiring modification of the input statements prior to running subsequent phases.)

Reevaluate your input graph definition statements and adjust as required. Specifying a Requires constraint where the subject node is not a primary cause may lead to unpredictable and undesirable test case output definitions.

***W25—Relation defining <nodename> moved internally to precede its first reference as a Cause***
Although desirable, it is not required that all relations statements be defined in their logical sequence of occurrence. BenderRBT has internally reordered the Relations statement for the named effect node since it was placed after another relations statement that referenced the named node as a cause.

***W26—ANCHOR node declaration for <nodename> ignored; not a Primary Cause***
The named node may have its state anchored using a Constraints statement only if it is a primary cause. (A node which appears to be a primary cause only within the scope of the cause-effect graph file in which it is defined, may in fact be an intermediate effect when the file is included as a subgraph within the scope of a master cause-effect graph file.) The named node does not have its state anchored as requested in the Constraints statement. This fact does not adversely effect other nodes named in the Anchor Constraints statement that are primary causes.

***W65—Undefined node name in TESTS statement: <nodename>***
The named node has been referenced in a Tests statement but not defined using a Nodes statement. The named node is not included in any subsequent test case definitions.

***W74—Subgraph filename expanded to: <fullFilenameSpec>***
The full device, directory and file name specification for a file referenced in a Subgraphs statement is displayed. Ensure that the device and directory name(s) displayed are the intended specification.

***W91—Ran 'Revise Descriptions" using input modified <date/time-stamp>***

For audit trail purposes only, the date and time that this file was subjected to the **Revise Descriptions** process is recorded in the Graph Errors report.

## 10.6 Functional Variation Report Messages

Interspersed throughout the Functional Variations listing may be any number of diagnostic messages. Any messages that appear having a **Vxx** identification code typically reports that a variation has been flagged as infeasible. Another series of diagnostic messages appearing in the Variations listing, as well as the Batch and Script Test Case Definitions, carry a **Txx** identification and are documented following this section.

Any **Vxx** messages that appear immediately precede the functional variation to which the message applies.

***Note:*** *When reviewing the list of functional variations produced, it is equally important to analyze those variations with and without diagnostic messages. Only you can be the judge as to whether each functional variation is consistent with the requirements specification.*

***V01—<Infeasible> Due to constraint(s) WITHIN relationship***
The effect state of the functional variation is not feasible given one or more constraints which have been applied to the cause nodes within the functional variation (i.e., at least one of the cause states in the functional variation has placed a constraint on one of the other cause nodes within the same functional variation). The functional variation will be excluded from subsequent **Design Tests** processing (i.e., the conditions present in the variation will not be tested). Verify that it is reasonable and consistent with the requirements specifications that this combination of node states be excluded from the test cases generated.

***V02—<Infeasible> ALL causes in this variation are MASKed***
All of the cause nodes in this variation are in a masked state; therefore, the state of the effect node cannot be determined.

The functional variation is excluded from subsequent **Design Tests** processing (i.e., the conditions present in the variation are not tested). Verify that it is reasonable and consistent with the requirements specifications that this combination of node states be excluded from the test cases generated.

***V03—<Infeasible> Effect state indeterminate after MASK(s) applied***

The state of the effect node, when taking into account the relational operator specified, cannot be determined due to the masking of one (or more) of the cause nodes. The functional variation is excluded from subsequent **Design Tests** processing (i.e., the conditions present in the variation are not tested). Verify that it is reasonable and consistent with the requirements specifications that this combination of node states be excluded from the test cases generated.

***V04—<Infeasible>Duplicates previous variation after MASK(s) applied***

One or more cause nodes in the variation have been masked. The remaining cause node(s) are in the same state(s) and the effect node is in the same state as is defined by a previous functional variation (which may or may not itself contain masked nodes). The functional variation is excluded from subsequent **Design Tests** processing (i.e., the conditions present in the variation are not tested). Verify that it is reasonable and consistent with the requirements specifications that this combination of node states be excluded from the test cases generated.

***V05—Due to constraint(s) ACROSS relationships***

The effect state of this functional variation is not feasible given one or more constraints that have been applied to the cause nodes of this functional variation. One or more constraints were applied based on the state(s) of one or more nodes in one or **V-**more preceding functional variations in a test case path (i.e., at least one of the cause states in the functional variation has

placed a constraint on one of the other cause nodes across two or more functional variations). The functional variation is excluded from subsequent **Design Tests** processing (i.e., the conditions present in the variation are not tested). Verify that it is reasonable and consistent with the requirements specifications that this combination of node states be excluded from the test cases generated.

***V06—Due to ANCHORed node constraint violation***
One or more nodes in the variation are primary cause nodes which have had their states declared to be held to true or false using an Anchor Constraint statement state. The functional variation is excluded from subsequent **Design Tests** processing (i.e., the conditions present in the variation are not tested). Verify that it is reasonable and consistent with the requirements specifications that this combination of node states be excluded from the test cases generated.

## 10.7 T-Diagnostic Messages in the Functional Variations Report

The **Txx** messages that appear immediately preceding a functional variation apply only to that Variation. **Txx** messages that appear at the end of a set of Functional Variations (and prior to the display of the next Relations statement in the listing) apply to the set of variations which precede the diagnostic message.

***Note:*** *When reviewing the list of functional variations produced, it is equally important to analyze those variations with and without diagnostic messages. Only you can be the judge as to whether each functional variation is consistent with the requirements specification.*

***T01—<Infeasible> Due to constraint(s) ACROSS relationships (or faulty logic)***
The effect state of this functional variation is not feasible given one or more constraints that have been applied to the cause nodes of this functional variation. One or more constraints were applied based on the state(s) of one or more nodes in one or more preceding functional variations in a test case path (i.e., at least one of the cause states in the functional variation has placed a constraint on one of the other cause nodes across two or more functional variations). Otherwise, “faulty graph specification logic” may sometimes be present; typically when two or more different-type constraints are applied to the same nodes within a variation. The functional variation will be excluded from subsequent **Design Tests** processing (i.e., the conditions present in the variation will not be tested). Verify that it is reasonable and consistent with the requirements specifications that this combination of node states be excluded from the test cases generated.

***T02—<Untestable> Due to non-OBServable effect(s) or unable to sensitize***

The state(s) of one or more cause nodes within the variation are not observable, therefore the state of the effect node cannot be validated (i.e., if there is a flaw in the logic associated with the non-observable cause state(s), then the true/false state represented by the faulty node cannot be relied upon).

Any intermediate effect nodes (synonymously referred to as cause nodes for the current functional variation) whose true/false state are verifiable during any test case scenario by virtue of their observability, should be annotated with the OBS designation in their node definition statements. Examples of observable nodes are: a field displayed on a terminal, a record data field whose contents can be printed or viewed after a test has been run; in short, any processing event that can be observed to have occurred, or conversely, to not have occurred. Any node which is not actually observable, but which you would like to *force* its observability to create a test case that includes this variation, should be defined using its node definition statement using the FOBS (forced) annotation.

***T03—<Untestable> Need to create EXPLICIT/OBServable intermediate node***

A non-observable, BenderRBT-generated intermediate node (i.e., the node name appears in the format **INT-xx*) has been created due to a compound Relations statement (i.e., one having two or more different relational operators, that has necessitated the need for enclosing two or more cause nodes in parentheses). Since the intermediate node is implied to exist, it is implicitly non-observable. If it is necessary to ensure that this functional variation is included in the test cases, then it is necessary to explicitly create an observable intermediate node and break the original **compound relationship** into two or more simpler relations statements.

***T04—<Untestable> Due to non-OBServable effect(s) or unable to sensitize***

BenderRBT was able to *extrapolate* the effect state of the functional variation but was unable to fully *sensitize* a path containing this variation in any of the test cases. This condition will either occur due to observability issues or a combination of node states being sensitized that does not match any of the functional variations defined. Examine the test cases generated. If it is necessary to ensure that this functional variation is included in the test cases, then it is necessary to either declare (or force) the observability of one or more of the cause states that precede this effect; or, define a test case (using a Tests statement) that covers the variation and reprocess the graph using the **Evaluate & Design BOTH** process.

***T05—<Not-tested> Not tested via OLD Test Case Definitions***
This message appears only when analysis of Old test cases (i.e., those defined using tests statements on the input graph file) is requested by the **Evaluate OLD** process. Any functional variation(s) associated with this message were not represented (i.e., covered) in any of the defined test cases. In analyzing the Old tests only, it has not been determined whether or not this variation is testable; it has only been determined that the variation is not covered in the test cases presented. If necessary, rerun BenderRBT and specify that both analysis of Old test cases and synthesis of New test cases be performed by requesting the **Evaluate & Design BOTH** process; any variations that are then infeasible or untestable are so marked.

***T07—Primary cause state for <nodename> in Old Test <testname> has been changed***
The primary cause state for the named node, that was declared using an Old Tests statement, has been changed in the named test case definition. Examine the test case definition generated. The logic defined using the Relations statements precludes using the primary cause node state specified. Verify that the logic is correct and that the corresponding primary cause node state in the test case definition is correct. Modify either the Relations statements or TESTS definitions, as appropriate.

***T08—<Infeasible> Due to constraint(s) ACROSS relationships (effect MASKed)***
The effect state of this functional variation is not feasible given one or more constraints that have been applied to the cause nodes of this functional variation. One or more constraints were applied based on the state(s) of one or more nodes in one or more preceding functional variations in a test case path (i.e., at least one of the cause states in the functional variation has placed a constraint on one of the other cause nodes across two or more functional variations). Further, the effect node state in this variation has been masked. An effect node is masked under one of two conditions: the effect node is named (specifically) as the object of a mask; or, all of the causes to the effect are currently in a masked state. The functional variation is

excluded from subsequent **Design Tests** processing (i.e., the conditions present in the variation are not tested). Verify that it is reasonable and consistent with the requirements specifications that this combination of node states be excluded from the test cases generated.

***T09—<Nodename> node state declared via OLD test <testname> was Masked***

The primary cause state for the named node, that was declared using an Old tests statement, has been masked in the named test case definition. Examine the test case definition generated. The mask condition defined using a Constraints statement now precludes using the primary cause node state specified. Verify that the mask and the logic are correct, and that the primary cause should indeed be masked. Modify either the Constraints statements or Tests definitions, if appropriate.

***T10—New primary cause state for <nodename> added to Old Test <testname>***

While processing an Old tests definition, a primary cause state not defined in the Tests statement has been added to the test case definition. Examine the test case definition generated to verify that it is correct and accurate. Normally, all of the primary cause states for a given Old test are specified using the Tests statement, although it is possible that the relations and/or constraints may have been modified or updated. This is an informational message only, assuming that the modified Relations and/or Constraints statements accurately represent the now current specification.

***T11—<Note!!> Probable graph logic error. TRUE/FALSE state of <nodename> always Infeasible***

Either TRUE or FALSE is indicated in the message. BenderRBT was unable to synthesize at least one test case specifying the true or false state of this variation's effect. Further, due to constraints imposed upon the graph, the true or false state of this effect is always infeasible. This is a probable (although not positive) indication of a logic error in the input graph specification. Review the input graph specifications and the output test cases created for completeness, accuracy and reasonableness. Especially review any multiple constraints affecting any cause nodes that precede this effect. If necessary, modify the input graph statements as required and reprocess this graph file.

***T12—<Note!!> TRUE/FALSE state of <nodename> not in any test case***
Either TRUE or FALSE is indicated in the message. BenderRBT was unable to specify the true or false state of this variation's effect in any test case, either through test case synthesis or through node state extrapolation. This is a probable (although not positive) indication of a logic error in the input graph specification. Review the input graph specifications and the output test cases created for completeness, accuracy and reasonableness. Review especially any multiple constraints affecting any cause nodes which precede this effect. If necessary, modify the input graph statements as required and reprocess this graph file.

***T13—<Note!!> TRUE/FALSE state of <nodename> in a test case but not fully sensitized***
Either TRUE or FALSE is indicated in the message. BenderRBT was able to *extrapolate* the true or false effect state of the functional variation but was unable to fully *sensitize* the effect state in any of the test cases created. This condition occurs due to observability issues or a combination of node states being synthesized that does not match any of the functional variations defined. Examine the test cases generated. If it is necessary to ensure that this functional variation is included in the test suite, then it will be necessary to either declare (or force) the observability of one or more of the cause states which precede this effect; or, define a test case (using a Tests statement) which covers the variation and reprocess the graph specifying that both Old test cases be analyzed and New test cases synthesized using the **Evaluate & Design BOTH** process.

***T14—<Note!!> There were NO TEST CASES generated!***
For some reason, there were no test cases generated during the Test Synthesis phase. Review the input graph specifications. Review the Relations statements given and the constraints imposed upon the graph. You will typically find that two or more constraint statements are in conflict with each other (i.e., they are establishing a logically impossible condition that prevents test synthesis from progressing through the graph). Modify the input graph statements as required and reprocess this graph file.

***T17—<Infeasible> Unable to sensitize the cause states***
BenderRBT was unable to include this variation in the suite of test cases, most likely due to the imposition of one or more

constraints statements upon one or more causes in the variation.
Examine the test cases generated. If it is necessary to ensure that this functional variation is included in the test suite, then it is necessary to either declare (or force) the observability of one or more of the cause states which precede this effect; or, define a test case (using a Tests statement) which covers the variation and reprocess the graph specifying that both Old test cases be analyzed and New test cases synthesized using the **Evaluate & Design BOTH** process.

***T18—<Infeasible> Due to intermediate non-OBServable effect(s) (or faulty logic)***
The effect state of this functional variation is not feasible given one or more constraints which have been applied to the cause nodes of this functional variation. One or more constraints were applied based on the state(s) of one or more nodes in one or more preceding functional variations in a test case path (i.e., at least one of the cause states in the functional variation has placed a constraint on one of the other cause nodes across two or more functional variations). Otherwise, "faulty graph specification logic" may sometimes be present; typically when two or more different-type constraints are applied to the same nodes within a variation. The functional variation is excluded from subsequent **Design Tests** processing (i.e., the conditions present in the variation will not be tested). Verify that it is reasonable and consistent with the requirements specifications that this combination of node states be excluded from the test cases generated.

***T21—<NOTE!!> LESS THAN 100% COVERAGE Achieved***
On completion of the Test Synthesis phase, when either the **Design NEW Tests** or **Evaluate & Design BOTH** process has been requested, BenderRBT was unable to achieve 100% coverage of the functional variations in the suite of tests generated. Review the cause-effect input graph specifications and the output test cases created for completeness, accuracy and reasonableness. Review especially any multiple constraints affecting the same cause nodes in the graph.

***T80—Note: Invalid/illogical/incomplete input definition; Test Case <testname> contains Indeterminate effect state(s)***
On completion of the Test Synthesis phase, when either the **Design NEW Tests** or **Evaluate & Design BOTH** process has

been requested, one or more effect states in the named test case have been declared Indeterminate. The presence of Indeterminate and/or blank effect nodes in any one test case in the suite generated by BenderRBT indicates that the test definition output should be considered invalid and that the cause-effect graph is either illogical or incomplete. The danger inherent in accepting any of the tests in this suite is that functional variations may be reported as *covered* when, in fact, they cannot be.

Review the cause-effect graph specifications and the output test cases created for completeness, accuracy and reasonableness. Review especially for any downstream effect nodes which are not masked, yet the upstream effect node is masked.

***T81—Note: Invalid/illogical input definition; Test Case <testname> contains Indeterminate effect states***

On completion of the Test Synthesis phase, when either the **Design NEW Tests** or **Evaluate & Design BOTH** process has been requested, BenderRBT encountered one or more effect nodes where a specific true, false or masked state could not be determined. (These nodes will appear in the Definition Matrix with an 'I' or 'i' indeterminate designation.) The presence of an Indeterminate effect node in any one test case in the suite generated by BenderRBT indicates that the test definition output should be considered invalid and that the cause-effect graph is either illogical or incomplete. The danger inherent in accepting any of the tests in this suite is that functional variations may be reported as *covered* when, in fact, they cannot be. An Indeterminate state occurs when, for example, all of the causes to an effect in an OR relationship are either false or masked, and at least one of the causes is masked. Another example is when all of the causes to an effect in an AND relationship are either true or masked, and at least one of the causes is masked. Typically, because a cause node has been masked, AND the state of any one of the other causes is not sufficient to determine the effect's state, then BenderRBT's only recourse is to flag the effect node as being indeterminate.

Review the cause-effect graph specifications and the output test cases created for completeness, accuracy and reasonableness. Review especially any multiple constraints affecting the same cause nodes in the graph, or for any constraint upon an intermediate effect that have precluded the sensitizing of the effect's state.

***T82—Note: Invalid/illogical input definition; Test Case <testname> contains one (or more) BLANK node states***

On completion of the Test Synthesis phase, when either the **Design NEW Tests** or **Evaluate & Design BOTH** process has been requested, BenderRBT was unable to establish the true, false or masked state of one or more effects in the named test case. The presence of one or more blank effect nodes in any one test case in the suite generated by BenderRBT indicates that the test definition output should be considered invalid and that the cause-effect graph is either illogical or incomplete. The danger inherent in accepting any of the tests in this suite is that functional variations may be reported as *covered* when, in fact, they cannot be. The most likely candidate situation leading to a blank effect node occurs when, in the course of designing the test case, the establishment of the logical true or false state of the effect node in question will violate some other relationship, or constraint, that is already established.

Review the cause-effect graph specifications and the output test cases created for completeness, accuracy and reasonableness. Review especially any multiple constraints affecting the same cause nodes in the graph or for any constraint upon an intermediate effect that have precluded the sensitizing of the effect's state.

# 11. Cause-Effect Graphing API

Prior to adding the graph drawing front (RBTg and previously via Visio), RBT used a Prolog based text front end. This is still used as an API to the Cause-Effect Graphing portion of the tool.

The Prolog language is the basis for the notation used in the cause-effect graphing language. The statements used in the two languages are compared in this table:

| Prolog | Cause-Effect |
|---|---|
| predicates/arguments | nodes |
| facts | constraints |
| rules | relations |
| goals | tests |

A few liberties are taken in the cause-effect language to simplify it for its specialized role, but these do not compromise the underlying compatibility between the two languages.

This compatibility is not accidental; both expert systems and cause-effect graphs describe behaviors, and both use declarative, non-procedural languages for this purpose. Similarly, we have developed a Writing Testable Requirements Style Guide which allows analysts to define their requirements in natural language (e.g. English, German) while still being unambiguous and deterministic. Therefore, the input to one also can be the input to the other. This commonality means that there is potentially a double payoff from writing functional specifications in a more formal style:

• Simulation of the program's behavior (using some Prolog compiler/interpreter) to verify that this behavior satisfies the user requirements for the program

• Analysis of the behavior and synthesis of the test specifications (using the BenderRBT system) to verify that this behavior is correctly implemented in the program's code

If both of these payoffs are realized, the cost of expressing the functional specification in a formal language is not borne by the program test activity alone, but is shared with the program design activity.

A number of Prolog compilers/interpreters are available, each one using its own variant of the Prolog language. The cause-effect graphing language is a close match to the generic Prolog language, though some minimal translation may be required to generate it from a particular Prolog variant.

## 11.1 Statement Types

A cause-effect graph (with or without a test library) is entered into BenderRBT using an ASCII text file. You may select the prefix of this file, but the file extension must be .CEG. This file consists of a stream of lines of text. Each line contains either a portion of or all of a statement in the cause-effect graphing language. One statement may span multiple lines, but no one line may contain multiple statements.

The following types of statements are used to define cause-effect graphs, as well as any existent tests of those graphs:

• Category Header statements:  Declares the type of statements that are to follow this statement. Statements that fall within any one category must be grouped together.

• Title statement:  Assigns a descriptive name to the current graph file. The contents of the title statement are placed on the first page of each printed output report.

• Node statements: Causes and effects are signified by node names. The list of these names and their expanded definitions appear in Node statements. The true or false state of one or more cause nodes in a relation, in combination with the relational operator used, determines the logical state of an effect node. An effect node which is both a cause and an effect within a graph is referred to as an intermediate effect node.

• Relation statements: These statements depict the logical relationships between causes and effects, as indicated by the boolean relational operators and, or, nand, nor, xor and xnor connecting the causes.

• Constraint statements: The boundary conditions which limit the invokable combinations of causes are delineated using Constraint statements.
• Test statements: The cause-state patterns of previously existing tests, if any, may be defined using these statements.

• Subgraph statements: A reference to another ASCII text file containing cause-effect graph statements may be included in the current input file using Subgraph statements.

## 11.2 Statement Syntax

The following conventions must be observed in BenderRBT statements:

• Category Headers, which are system-defined, may be entered using any combination of uppercase or lowercase letters, but may not be abbreviated. In this chapter, Category Header statements are shown in all uppercase letters.

• Keywords, which are system-defined, may be entered using any combination of uppercase or lowercase letters, but may not be abbreviated. In this chapter, statement keywords are shown in all uppercase letters.

• Node names, Test names and File names, which are user-defined, may consist of any mix of these characters: A through Z, a through z, 0 through 9 and the seventeen characters: ! @ # $ % ^ - _ ? \ " & + < > { }
The following fifteen characters may not be used: ( ) [ ] . , ; : | / ' * ` = ~

Note that uppercase and lowercase letters may or may not be distinctive, depending upon the current setting of the Case Sensitivity check box in the Run menu.

Node names should not be used as Test names and vice-versa. As BenderRBT may be unable to distinguish between the two uses, unpredictable results will occur.
References to example Node names, Test names and File names appear in this chapter.

• Each statement whose syntax definition appears in this manual enclosed within brackets ( [ ] ), may or may not contain the enclosed term(s), at the discretion of the user.

• Each statement whose syntax definition appears in this manual following an ellipsis (...), may have the term preceding the ellipsis repeated any number of times.

• One or more blanks are required as delimiters between terms unless some punctuation mark is specified in the syntax definition as the delimiter.

• The number of characters contributing to statement length is tallied as follows: Any number of blanks is allowed at the beginning of a statement to permit indentation and are not counted. Two or more blanks appearing between terms in a statement are counted as one character. Multiple blanks appearing within a term in a statement (e.g., a node description field) are all retained and counted as individual characters. The period at the end of each statement is counted as one character and serves to reset the statement length counter. Any carriage return and line feed characters embedded within an ASCII text file and used as logical record delimiters are only counted as one character when they occur between terms in a single statement.

• Any tab character (ASCII value 09) embedded within a statement will be converted to a single blank character. Note that when a text editor other than the facility provided by BenderRBT is used to create or modify a graph file, the tabbing conventions used by the other editor may not coincide with those used by BenderRBT and therefore may not always be aligned in the desired format.

• Every statement (except Category Headers) must be terminated by a period. Periods may not appear within a statement, except within comments.

• An input line may contain up to 250 characters. A statement may consist of multiple lines, but the total of all characters in a multiple line statement cannot exceed 4,090 characters. (Blanks and tabs used for indentation or padding, as well as carriage return and line feed characters, contribute to input line length, but do not contribute to statement length.)

• Comments may appear anywhere within the series of statements which define a graph. Any comment preceded by the characters /* must be followed by the characters */. A comment which is delimited by the /* and */ pair may span multiple lines. A comment preceded by the characters // is terminated at the end of the current text line, which means that it must be contained in whole within that one line. A comment may appear on a line by itself or placed at the end of a normal statement line. Comments are ignored by BenderRBT and do not contribute to statement length.

## *11.3 Title Statement*

The Title statement assigns a descriptive name to a particular graph.

A Title statement should be the initial statement in the stream of statements that define a graph. For each graph, only the first Title statement is used; any subsequent ones are ignored. This permits inclusion of subgraphs without overriding the title of the parent graph.

The format of the Title Statement is as shown below:

```
TITLE   [ 'description' ]  .
```

The parts of the statement are:

TITLE: The keyword TITLE must be the first word in the statement.

'description': The optional words that follow TITLE constitute the description that begins all of the reports generated by BenderRBT for a given graph. Although the description may be up to 250 characters, adherence to a more practical limit of 60 characters or less results in a single title line description appearing at the top of each printed report. The description must be enclosed by a pair of single quotation marks. Any ASCII characters except single quotes may be used within the description. Finally, the description and its enclosing quotation marks must be contained in whole within a single input line.

Title statement examples:

```
TITLE 'SAVE command'.
TITLE.
```

## 11.4 Nodes Statements

The Node statements define the nodes that are explicitly depicted in a graph.

The header for the Nodes section should be formatted as shown below:

NODES

Node statements must be grouped together, with the NODES header being the first one in the group. Multiple groups of Node statements are permitted as long as each group is led by its particular Nodes header.

The format of the Nodes statement is as show below:

Case 1:
nodename [ = 'true-state description' ] [OBS/FOBS]

Case 2:
nodename [ = 'true-state description' [ | ] [ /B ] ] [OBS/FOBS]

Case 3:
nodename [ = 'true-state description' [ | 'false-state description' ] ] [OBS/FOBS]

The parts of the statement are:

nodename: This is the shorthand name given to the node. It is composed of between 1and 32 alphanumeric characters. More than 32 characters results in a Severe error [S32], which terminates further processing. It is recommended that node names be held to a maximum of 15 characters, primarily for ease (and accuracy) of data entry.

The last Node statement within the input stream containing a given node name is accepted as the definition of that node. Any previous Node statement with the same node name is ignored, and a warning message [W03] is generated.

Note: The following headers and keywords, in any combination of upper and lowercase characters, may not be used as node names:
TITLE, NODES, RELATIONS, CONSTRAINTS, TESTS, SUBGRAPHS, OBS, FOBS, NOBS, PAS, EXCL, INCL, ONE, REQ, MASK, ANCHOR, AND, OR , NOT, NAND, NOR, XOR, XNOR.
Node names declared using any of these words are not accepted by BenderRBT.

A node name must be defined if it is used in a Relations statement. If it is not, then the Relations statement is rejected, a Severe error message generated and further processing terminated.

A node name must be used in a Relations statement to also be referenced in a Constraints statement. The Relations statement referencing the node name may appear before or after the Constraints statement. If a Constraints statement references a node name not used in a Relations statement, then the Constraints statement is rejected.

The sequence in which node names appear in test case descriptions is primarily based upon their sequence of definition within Relations statements, not based upon their sequence of definition using Nodes statements. The Relations statement for each effect node should appear in sequence ahead of any subsequent Relations statements that reference the same effect node as a cause node. (Refer to the topic Relations Statements later in this section for a further clarification of this rule.)

True-state, false-state description: The meanings of the true and false states of a node may each be separately specified in a free-form description enclosed by a pair of single quotation marks. Any ASCII characters except single quotes may be used within the description.

Any primary cause node must have at least a true-state or a false-state description defined that is other than null or /B.

Except for primary cause nodes, node name descriptions are optional; however, if any description is provided, then an equal sign (=) must separate the node name from the true-state description provided.

If both descriptions are completely omitted, then the node's name is used for the true-state description, and the NOT nodename sequence used for the false-state description.

A null description may be indicated by using two consecutive single-apostrophes (e.g., Nodename = ‘’.) and this circumvents BenderRBT's substitution of the node name for the description.

A node's true and false-state descriptions may each contain as many as 4,090 characters. If it is longer, a Severe error message is generated and further processing terminated. It is recommended that descriptions be held to a maximum of 60 characters (i.e., one line of print).

If you must declare descriptions which exceed 60 characters, then it is necessary (and desirable) to span statement lines in order to successfully declare the long description string. To span multiple statement lines when declaring a long node description, do not terminate each statement line with a closing apostrophe. Simply terminate the line with a carriage return and continue the description on the next line. Note that any leading white space, if any, on the continued line is included in the description.

When specified, the false-state description must be separated from the true-state description by the vertical-bar ( | ) character.

When a true-state description is provided and a false-state description is not provided, BenderRBT prefixes any references to the false-state condition with the word NOT in front of the true-state description provided, unless the /B option (see below) is specified.

When a null true-state description is provided (i.e., two consecutive single apostrophes are used) and a false-state description is provided, and references in the test case listings to the true state of a node are suppressed. In this case, only the false state descriptions provided appear in the test case listings.

/B: BenderRBT-generated references to any false-state conditions for a node name appearing in the test case listings may be suppressed entirely by specifying the /B (bypass or blank) option instead of a false-state description. A lowercase /b specification produces the same results, regardless of the current case sensitivity setting.

Note: In any test case definition where all of the nodes named in an exclusive Constraint are specified to be in their false state, BenderRBT overrides any /B declarations present. In this situation, BenderRBT prefixes the true-state description with the word NOT for each of the nodes named in the Exclusive Constraint.

OBS: Primary causes (nodes that are not also effects) and primary effects (nodes that are not also causes) are implicitly observable.

Intermediate effects (nodes which are both a cause and an effect) may be explicitly designated as observable effects by specifying the OBS parameter. An intermediate node should be flagged as observable only when the node represents an effect which can be observed, such as: data displayed on a screen, an update to a database than can be verified, output printed on a report or a packet being sent in a communications program.

FOBS: Intermediate nodes that are not normally observable effects, but must somehow be represented as observable in order to test the system's logic, may be designated as a forced-observable effect by specifying the FOBS parameter. In essence, these nodes denote where diagnostic probes should be built into the software to ensure full testing of the function.

Primary effects should be explicitly designated as observable only if all of the following conditions exist:
• The subject graph is itself a subgraph of another graph.
• The subject effects are directly observable in the context of the integrated graph.
• The subject effects are intermediate nodes in the integrated graph.

Note: For a complete discussion of true- and false-state descriptions and observable vs. forced-observable considerations, refer to the section titled OBS/FOBS Declaration in Node Definition in Chapter 4 of the User Tutorial.

Nodes Statement Examples:

```
NODES
Filename1 = ‘A valid filename is specified’.
Filename2 = ‘A valid filename is specified’
                | ‘An INVALID filename is specified’.
Filename3 = ‘Valid filename’ | ‘INVALID filename’ OBS.
newname1 = ‘specified(new_name)’.
newname2 = ‘specified(new_name)’ | /B.
newname3 = ‘specified(new_name)’ /B.
oldname    =  ‘specified(old_name)’.
SaveSpec  =   ‘File is saved under the specified name’ obs.
SaveDef     =   ‘saved(file,default_name)’ Obs.
MemSame1 = ‘unchanged(“memory and display”)’.
MemSame2 = ‘unchanged(“memory and display”) FOBS.
anyNode1=.
anyNode2 FOBS.
IntNode=’ ‘ | /B.
QuickNode.
```

A Data Dictionary of node names used for a group or library of cause-effect graph files may be created and maintained as a single input graph file. This file should contain only:

• A NODES header statement.
• Any number of nodename definition statements.
• Any optional comment statements.

The name of this Data Dictionary graph file should be included using a SUBGRAPHS declaration placed immediately after the Title statement in a main graph file.

For example: Create a graph input file named NODE_LIB.CEG and place the following statements in it:

```
/* NODE_LIB.CEG  -  Data Dictionary of Nodenames */
/* Created by:      A. User                  */
/* Last Modified:   1/15/92 by AU              */
NODES
Name = ‘Valid name entered’              OBS.
Addr1 = ‘Valid address line-1 entered’      OBS.
Addr2 = ‘Valid address line-2 entered’      OBS.
City  = ‘Valid city entered’ OBS.
State = ‘Valid state abbreviation entered’  OBS.
:
:
```

Then, each main graph file which uses the Data Dictionary should begin with the following statements:

```
TITLE 'Sample Data Dictionary Reference'.
/* SAMPLE.CEG  -  Sample Data Dictionary Reference  */
/* Created by:    A. User                    */
/* Last Modified: 6/1/93                      */
SUBGRAPHS
   \BenderRBT\proj01\NODE_LIB.
RELATIONS

:
:
[etc.]
:
:
```

If it is necessary to override the Data Dictionary's definition of a node name (for example, you may wish to override a node's false-state description with a /B [bypass false-states] or add an OBS [observable] designation), then simply redefine the node name by placing a new node definition statement after the Subgraphs' reference to the Data Dictionary's file name.

In effect, the last definition encountered in the graph file for a node name is the definition that BenderRBT uses. A Warning message is placed in the Graph Entry error listing for audit trail and verification purposes.

The number of node name definitions retained in a Data Dictionary file is limited only by the amount of hard-disk space available to store the file. See Chapter 7: System Limits to determine the maximum number of node names which may be referenced (using Relations statements) in any one graph file.

Constraint statements should not be included in a Data Dictionary subgraph, as it is highly unlikely that all of the nodes specified in the constraint are referenced in Relations statements in the main graph file. It is suggested that applicable constraints be included as Comment statements in the Data Dictionary as reminders of their applicability when the node name definitions are utilized by the main graph file. In short, any node name referenced in a Constraint statement and not referenced in a Relations statement causes a Severe error message to be generated and further processing terminated.

## *11.5 Constraints Statements*

The constraint statements specify boundary conditions of the graph which directly or indirectly limit the combinations of cause states that can be invoked.

The header for the Constraints statements is as shown below:

CONSTRAINTS

The Constraint statements must be grouped together with the CONSTRAINTS header leading the group. Multiple groups of Constraint statements are permitted provided that each group is preceded by a Constraints header.

The format of the Constraints is as shown below:

MASK ([NOT] subjectNode, objectNode [, ... ] ) .

If the leading subjectNode (the subject of the Mask statement) is in the specified state, then the state(s) of the succeeding objectNode(s) (the objects of the Mask) are undefinable. Common synonyms which may be substituted for undefinable are indeterminate and irrelevant.

Note: The state(s) of the succeeding object nodes are irrelevant, and therefore the false state of any object nodes may not be specified using the NOT keyword in the Mask statement. For example, the Constraint statement “MASK(X, NOT Y)” is illogical and not allowed.

REQ ([NOT] subjectNode, [NOT] objectNode [, ... ] ) .

If the leading subjectNode (the subject of the Requires statement) is in the specified state, then the succeeding objectNode(s) (the objects of the requirement) must be in their specified state(s).

EXCL ([NOT] node, [NOT] node [,[NOT] node ... ] ).

The specified nodes are mutually-exclusive, meaning that at most one of the nodes in the set may be in the specified state in each test.

INCL ([NOT] node, [NOT] node [,[NOT] node ... ] ).

The specified nodes are all-inclusive, meaning that at least one of the nodes in the set must be in the specified state in each test.

ONE ([NOT] node, [NOT] node [,[NOT] node ... ] ).

The specified nodes are both mutually-exclusive and all-inclusive, meaning that one and only one of the nodes in the set must be in the specified state in each test.

ANCHOR ([NOT] node [, [NOT] node, [NOT] node ... ] ).

If any of the specified nodes are a primary cause node within the context of the current graph file, then those primary cause nodes will be held to the specified state in each test. If any of the specified nodes are not primary causes, as may be the case when a graph has been included as a subgraph within another graph file, then the declared Anchored states of the non-primary cause nodes will be ignored.

The parts of the Constraint statements are:

node: Each node is identified by its node name, which may or may not be preceded by a NOT connective. If the NOT is present, then the node state is false in the constraint. If the NOT is omitted, then the node state is true in the constraint.
The true and false states of any one node may not both be specified in any one or more Constraint statements, as the two states are mutually exclusive.

All node names used in Constraint statements must also be referenced in Relations statements (and defined using a Nodes statement).

BenderRBT issues a Severe error message for any constraint statement using a node name that is also not referenced in a Relations statement.

In general, constraints should specify only primary causes. This follows from the definition of constraints as environmental limitations on the feasible combinations of input boundary conditions.

However, constraints may also specify intermediate nodes and even primary effects as indirect limitations (using the intervening graph logic) on the feasible combinations of primary causes when sets of input causes are being constrained. For example, consider the following (partial) graph statements:

```
Relations
X :- A or B or C.
Y :- 1 or 2 or 3.

Constraints
EXCL(A,B,C).
EXCL(1,2,3).
ONE(X,Y).
```

In this scenario, ONE (and only one) member of the A,B,C set or the 1,2,3 set must be present.

This does not imply that the states of the non-invokable nodes themselves are being constrained; as the states of these nodes are determined solely by the logic in the graph, subject to the possibility that this logic is flawed in the implementation of the graph.

This use of non-invokable nodes in constraints effectively buries connectives within the constraints, thereby permitting more complex constraint expressions. If the connective structure needed in a constraint does not already exist in the graph, it can be defined using a relations statement carrying the PAS annotation. This excludes it from the graph being tested, but includes it in the constraint which specifies it.

Note: In short, do not constrain intermediate nodes based on the output results expected; constraints should be imposed solely on the basis of the input boundary conditions present.

When synthesizing a test path involving variations containing masked nodes, BenderRBT may extrapolate the true/false state of an effect when a sensitized variation exists involving one or more masked causes, only when the true- or false-state of any one remaining cause is sufficient to determine the state of the effect after taking into account the relational operator used. For example, in synthesizing a test case involving the relation:

X :- A and B and C

If B and C are both in the masked state, and if A is false, then (because of the AND relational operator) it can be deduced that X is false. Alternatively, if B and C are both in the masked state, and if A is true, then the state of the X effect cannot be logically determined and therefore is not sensitized. The similar extrapolation of an effect state occurs for the other relational operators only when the state can be logically deduced.

When synthesizing a test path involving variations containing masked nodes, if all of the causes in a sensitized variation are in a masked state, then the effect node of the variation is also set to the masked state.

When synthesizing any test path, Constraint statements are evaluated and applied in the following sequence: Masks, Requires, Exclusives, Inclusives, Ones and Anchors.

When one or more (but not all) of the nodes named in a One or Inclusive constraint are masked, then the remaining (i.e., non-masked) nodes must satisfy the One or Inclusive constraint.
Similarly, none of the object nodes named in a Requires constraint may be masked when the subject state of the Requires constraint exists.

Constraints Statement Examples:

```
CONSTRAINTS
one (newname, oldname).
MASK (NOT Filename,newname,oldname).
EXCL(EscKey,CarrRtn,PageUp,PageDn,HomeKey,EndKey).
INCL(Addr1,Addr2,Addr3).
Req(ErrorPrompt,CarrRtn).
```

ANCHOR (Begin).

Caution: Constraints are the trickiest statements to use, particularly when connectives are effectively buried within them. In this situation, use care to avoid circular logic. This occurs when one can trace from an object node specified in any one constraint back through the graph (and possibly through other constraints) to the original constraint. Be especially wary of two or more constraints of differing types (e.g., a Mask and an Exclusive) that name common nodes. If circular logic is present, its impact on the execution of BenderRBT is unpredictable. The most common indicators of the presence of circular logic are the following:
• A small number of test cases with a high number of Infeasible and/or Untestable variations.
• Test cases which have not been extended through to a primary effect node.

## *11.6 Relations Statements*

The Relations statements express the logical relationships between the causes and effects in the body of the graph. Each statement links one or more explicitly-named causes to a single explicitly-named effect.

The header for the Relations section should be formatted as shown below:

RELATIONS

The RELATIONS header must appear at the beginning of a group of Relation statements. There may be multiple groups of Relations statements, but each group must start with a Relations header.

The format of the Relations statement is as shown below:

```
   effect  :-  cause-construct  [PAS] .
```

The parts of the statement are:

effect: This is the node name of the single effect in the relation statement. A NOT connective is not permitted ahead of this node name.

:- This symbol separates the effect from the cause-construct. It is equivalent to the logical “if and only if”. This symbol consists of the colon and hyphen characters, with no intervening space characters.

cause-construct: This is composed of one or more causes, and zero or more logical connectives, structured in one of the following ways:
• Cause
• Not cause
• Cause connective cause

Any cause shown in these cause-constructs may, in fact, be a lower-level cause-construct.

The eligible logical connective names and their meanings are:

NOT: Logical negation, which specifies that the effect is true if the cause-construct that succeeds it is false, and that the effect is false if the cause-construct which succeeds it is true.

AND: Logical conjunction, which specifies that the effect is true if and only if both of the cause-constructs which flank it are true; otherwise, the effect is false.

NAND: Logical conjunction followed by logical negation, which specifies that the effect is false if and only if both of the cause-constructs which flank it are true; otherwise, the effect is true.

OR: Logical disjunction, which specifies that the effect is false if and only if both of the cause-constructs that flank it are false; otherwise, the effect is true.

NOR: Logical disjunction followed by logical negation, which specifies that the effect is true if and only if both of the cause-constructs that flank it is false; otherwise, the effect is false.

XOR: Logical disjunction, which specifies that the effect is true if one and only one of the cause-constructs that flank it is true; otherwise, the effect is false.

XNOR: Logical disjunction, which specifies that the effect is true if one and only one of the cause-constructs which flank it is false; otherwise, the effect is false.

Cause-constructs are evaluated left to right; the NOT connective takes precedence over all other connectives. Parentheses must be used to delineate the causes and their related connectives any time there is a change in the relational operator used; which is to say that there is no operator precedence implicit within the cause-effect graphing language. For example, the relationship

X :- A OR B AND C is potentially ambiguous; therefore, it must be qualified using parentheses, as in X :- (A OR B) AND C , or X :- A OR (B AND C)

Superfluous or unnecessary parentheses are not accepted by BenderRBT on the basis that it may indicate the presence of a partially formed relations statement. For example, the following statement (although logical) is not accepted:

X :- (A and B and C and D)

Note: Relations statements are evaluated from top to bottom (i.e., in the sequence in which they are defined) and an internal list of node name references is constructed based on a logical sequencing of cause and effect nodes. It is considered to be good statement coding practice to declare your Relations statements in a logical sequence (i.e., an effect node name declared in any one relations statement should appear before the same nodename is again referenced as a cause node in a subsequent relations statement). Note, however, that BenderRBT resequences its internal list of node names as required to adhere to the requirement of the foregoing sentence.

A given cause may appear more than once in the same statement when more than one relational operator is used. For example, the following statement is acceptable:

XYZ :- (A and B) or (C and not B) or (D and B)

Node names used in relation statements to identify causes and effects must have been defined as a Nodes statement somewhere in the graph. BenderRBT also rejects any Relations statements using an undefined node name.

The sequence in which node names appear in test case descriptions is based upon their sequence of definition within Relations statements, not based upon their sequence of definition using Nodes statements. Further, within each Relations statement, the cause nodes are evaluated before the effect nodes. For example, consider the following Relations statements:

X :- A or B or C

Y :- D or E or F

In this case, the nodes appear in the following priority sequence within any test cases which reference them:

A, B, C, X, D, E, F, Y

PAS: Relations within the scope of the test effort are active and those outside this scope are passive. Passive relations are used to define “scaffolding” used in testing the active relations. Such passive relations are not addressed by the functional variations or test cases. Relations are presumed active unless explicitly designated as passive by the PAS parameter.

Relations Statement Examples:

```
RELATIONS
SaveDef :- NOT filename.
```

SaveSpec:-Filename and (newname OR oldname) PAS.
MemSame :- SaveDef or SaveSpec.

## *11.7 Test Statements*

Test statements describe the cause states of existent tests. When applicable, these statements are included with the graph definition file to which they apply.

The header for the Test section should be formatted as shown below:

TESTS

Test statements must be grouped together with the TESTS header preceding the group. Multiple groups of test statements are permitted as long as each group begins with a Tests header.

The format of the Test statement is as shown below:

[testname =] [NOT] cause [, [NOT] cause ...  ] .

The parts of the statement are:

testname:   Each test may or may not be uniquely identified by a user-assigned name. If specified, the name is limited to 32 alphanumeric characters; a test name longer than 32 characters results in a Severe message that terminates further graph processing.

The following headers and keywords may not be used as test names:
TITLE, NODES, RELATIONS, CONSTRAINTS, TESTS, SUBGRAPHS, OBS, FOBS, NOBS, PAS, EXCL, INCL, ONE, REQ, MASK, ANCHOR, AND, OR, NOT, NAND, NOR, XOR, XNOR.

In addition, any defined node name may not be used as a test name. A Test statement having any of these names causes unpredictable results when evaluating the remainder of the graph file.

When specified, the test name must be followed by an equal sign (=) to clearly delimit it from the cause states.

Cause:   Each cause is identified by its node name, which may or may not be preceded by a NOT connective. If the NOT is present, then the cause state is declared as false in the test. If the NOT is omitted, then the cause state is declared as true in the test.

In general, node names should have been defined previously in Node statements and referenced in a Relations statement; however, some node names may fall outside the scope of the subject graph and yet be valid names in the context of some corollary graph.

BenderRBT issues a warning message listing any undefined node names in a Test statement, but does not reject the statement.

Each cause must be directly invokable-that is, it must be a primary cause since intermediate nodes are not directly invokable. BenderRBT rejects any Test statement that specifies a non-invokable cause.

In a hierarchy of graphs, Test statements typically are appended to the graph at the top of the hierarchy, though they could appear at any level. In any case, all old tests are applied to the composite graph, not just to the graph in which they are defined. Thus, the restriction on specifying only directly-invokable causes in Test statements is enforced using the primary causes of the composite graph.

Test Statement Examples:

```
TESTS
FirstName, not Mid_Init, LastName.
mytest = FirstName,Mid_Init,NOT LastName.
NOT LastName.
```

## 11.8 Subgraph Statements

Subgraph statements identify the object graphs to be integrated with the subject graph.

The header for the Subgraph section should be formatted as shown below:

SUBGRAPHS

Subgraph statements must be grouped together following the SUBGRAPHS header. Multiple groups of Subgraph statements are permitted as long as each group is preceded by a Subgraphs header.

The format of the Subgraph statement is as shown below:

```
fileSpec [ PAS ] .
```

The parts of the statement are:

fileSpec: Each subgraph is identified by its file name specification. If the complete drive designation and directory name(s) are not specified, then the drive designation and directory name(s) of the cause-effect graph file is used. Do not include the Subgraph's ".CEG" file name extension in the fileSpec.

Note: The file extension is implicitly .CEG and can not be specified explicitly since periods are reserved for use as statement terminators.

PAS: A subgraph may be active (within the scope of the test effort) or passive (outside the scope of the test effort). Passive subgraphs serve as scaffolding for testing the subject graph. Unless explicitly designated as passive by the PAS parameter, a subgraph is considered active.

Subgraphs may in turn point to still lower-level subgraphs. The only limit on the number of levels in this hierarchy is established by the operating system's limit on the number of concurrently open files (see Chapter 7: System Limits) and available RAM. A given subgraph may be pointed to by multiple higher-level graphs.

The rule that only previously-defined nodes can be referenced by Relation and Constraint statements also applies when such statements straddle multiple graphs. Since each subgraph is processed when the first Subgraph statement pointing to it is encountered, the order in which Subgraph statements are executed can prove critical. Therefore, you must carefully locate subgraph statements in the hierarchy of graphs.

Refer to Maintaining a Data Dictionary of Defined Node Names on page 20 regarding the use of a Subgraphs statement to declare and use a Data Dictionary library of node name definitions.

Subgraphs Statement Examples:

```
SUBGRAPHS
graph12   PAS.
C:\subjspec\custblg\newadd.
```

# 12. Additional RBT Features Not Available Via RBTg

There are a few features that are available only through the API and cannot yet be accessed via the drawing facility in RBTg. These are Subgraphs and Passive. Another feature that can only be accessed via RBT, but not via RBTg, is setting up and running a Queue of graphs.

## *12.1 Subgraphs*

Subgraphs allows you to break a large graph into multiple small graphs and then merge them together in the same way Include statements might be used in a programming language. To ensure that the pieces link up properly you need to ensure that the node names match up across the graphs. See the chapter on the API to see the syntax of the subgraphs statement.

One use of subgraphs is to have a master set of common nodes. Then all of related graphs would include this subgraph to ensure that a given node's description was identical across all of the graphs. A way to do this in RBTg is to create a graph file which only contains nodes and their definitions. When you start a new graph you just copy over the needed nodes and paste them in.

## *12.2 Passive*

Passive is an attribute (using PAS) that can be applied to a subgraph or a relations statement. It allows you to include logic in your graph that does need to be tested but through which other logic must pass. RBT will not generate a full set of functional variations for anything marked PAS. For example, if you had an OR relationship with five causes it would generate six variations. RBT would ensure that all six were in one or more tests. However, if it was marked PAS it might only need to use two – one resulting in a true effect and another resulting in a false effect – to test active relations upstream and downstream in the logic.

The use of Passive is consistent with testing at multiple levels – e.g. unit test, component test. You might have thoroughly tested the rules represented in a subgraph at the unit level. You would not have to test each variation again at the component level. You just want to make sure that the "handshake" between the units is working and that the overall flow of the logic works across the units – i.e. intra-unit testing versus inter-unit testing.

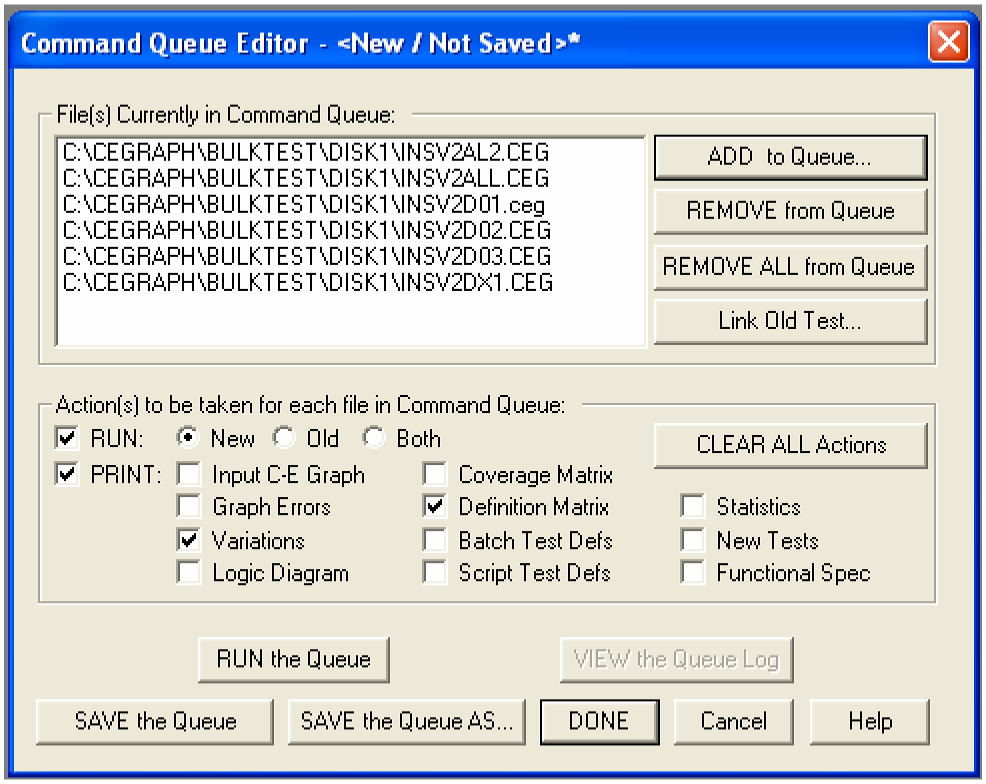
## *12.3 Command Queue*

The Command Queue facility accommodates the running and printing of one or more cause-effect graph (.ceg) files in an unattended (batch processing) mode. You cannot use it to run multiple .rbt files. This feature was originally built into RBT to allow us to test the test case design engine and the various reports. It turned out to be useful in day to day work also when we have graphs that take a long time to run. We just set up a queue of them and let them run over night. Even if you are using the RBTg front end, this can still be useful. If you have a longer running graph you can cancel the run. The .ceg file has still been created. You can then run the graph directly in RBT. If you have a number of them you can set up a Queue and run them all.

When you select the File > Command Queue menu item, two menu items are presented: New Queue and Open Queue.

Select File > Command Queue > New Queue to create a new Command Queue file.
Select File > Command Queue > Open Queue to access a previously created command queue file.

In both cases, if there was a previously open cause-effect graph file, the Graph Editor and related reports are closed. Then the Command Queue Editor dialog box appears:



File(s) Currently in Command Queue:

```
C:\CEGRAPH\BULKTEST\DISK1\INSV2AL2.CEG
C:\CEGRAPH\BULKTEST\DISK1\INSV2ALL.CEG
C:\CEGRAPH\BULKTEST\DISK1\INSV2D01.ceg
C:\CEGRAPH\BULKTEST\DISK1\INSV2D02.CEG
C:\CEGRAPH\BULKTEST\DISK1\INSV2D03.CEG
C:\CEGRAPH\BULKTEST\DISK1\INSV2DX1.CEG
```

Action(s) to be taken for each file in Command Queue:

✓ RUN: [✓ New] Old Both

✓ PRINT: Input C-E Graph; Graph Errors; ✓ Variations; Logic Diagram; Coverage Matrix; ✓ Definition Matrix; Batch Test Defs; Script Test Defs; Statistics; New Tests; Functional Spec

**Command Queue Editor**

There are two primary tasks to be accomplished before executing the Command Queue: building the list of .ceg files to be included and specifying the desired run and/or print options.

**Command Queue Editor: Building the List of Files**

**ADD to Queue** Button: On selecting the ADD to Queue button, a standard File > Open dialog box is presented, which may be navigated to select one or more cause-effect graph files to be processed. Cause-effect graph files may be selected from this dialog box only one subdirectory at a time, although the command queue list of files to be processed may consist of files from multiple devices and directories.

To select multiple cause-effect graph files from the same directory, either hold down the Ctrl (Control) key while selecting individual files or hold down the Shift key while selecting the first and last files in a range of contiguous file names you wish to process.

When one or more files in a directory have been selected, press the ADD button in the File > Open dialog box to return to the Command Queue Editor dialog box. The file(s) selected are added to the end of the command queue file list, not sorted or merged into the list.

To return to the command queue editor without making any selections from the File > Open dialog, press the Cancel button.

**REMOVE from Queue** Button: One or more of the cause-effect graph files in the list of files to be processed may be removed from the command queue by selecting them and then pressing the REMOVE from Queue button.

To select multiple files in the list, either hold down the Ctrl (Control) key while selecting individual files or hold down the Shift key while selecting the first and last files in a range of contiguous file names you wish to process.

**REMOVE ALL** from Queue Button: To remove all of the cause-effect graph files from the list of files to be processed, simply press the REMOVE ALL from Queue button.

**Specifying the Run/Print Action**

**CLEAR ALL** Actions Button: All of the check marks associated with the print action are cleared, the Print Action check box is cleared, and the Run Action defaults to New when the CLEAR ALL Actions button is selected.

**RUN** Action Check Box: If the Run Action check box is selected, then each of the cause-effect graph files appearing in the list of files to be processed are run using the New/Old/Both radio button setting specified.

**PRINT** Action Check Box: If the Print Action check box is selected, one or more check boxes for each of the reports to be printed should also be selected. When the Print Action check box is selected, the selected reports are printed for each of the cause-effect graph files appearing in the list of files to be processed.

**Dialog Buttons**

**RUN** the Queue Button: The RUN the Queue button is only active when both of the following conditions exist: one or more files appear in the list of file names to be processed; and the Run and/or Print Action boxes are selected.

On selecting the RUN the Queue button, a copy of the file names in the list box and the specified actions is saved in a command queue file having a filename suffix of .QUE. If this is a new command queue file, then a File > Save As dialog box is presented so you may specify the name and desired location of the file. The run and/or print actions specified are then carried out for each of the cause-effect graph files listed. No other BenderRBT-related processing (such as editing a cause-effect graph file) may be undertaken while the Command Queue processor is running.

During the running of a command queue, the Thermometer dialog is displayed, indicating the cause-effect graph file currently being processed and its status.

Note: The use of an animated screen saver during a lengthy command queue process greatly impedes BenderRBT’s ability to efficiently generate test cases in a reasonable amount of time. BenderRBT’s Run process is a CPU-intensive application. Any and all of the system’s available processing power is consumed during this period. Sharing the system’s computing resources with another application, especially an animated screen saver, only lengthens the amount of time required to produce BenderRBT output. It is highly recommended that your screen saver utility be deactivated, or set to blank-screen, during any extended command queue processing.

At the completion of a request to RUN the Queue, a Command Queue Log is displayed. An indication of the success or failure of each of the actions requested for each of the files is printed in the Log, as well as time-stamps and run-time summaries. The Command Queue Log file has a file name consisting of the prefix portion of the input .QUE file name, followed by the suffix .LOG.

**VIEW** the Queue Log Button: The contents of the previously created Command Queue Log file associated with the current queue are displayed by pressing the VIEW the Queue Log button. If there is no Log file available, this button is disabled.

**SAVE the Queue** Button: The SAVE the Queue button is only active when both of the following conditions exist: one or more files appear in the list of file names to be processed, and the Run and/or Print Action boxes are selected.

The SAVE the Queue button saves a copy of the file names in the list box and the specified actions in a command queue file having a file name suffix of .QUE. If this is a new command queue file, then a Save As dialog box is presented so you may specify the name and desired location of the file.

**SAVE the Queue As** Button: The SAVE the Queue As button is only active when both of the following conditions exist: one or more files appear in the list of file names to be processed and the Run and/or Print Action boxes are selected.

The SAVE the Queue As button may be pressed to create a new copy of the current command queue file. A Save As dialog box is presented so you may specify the name and desired location of the file.

**DONE** Button: Selecting the DONE button closes the command queue dialog box when you are finished working on the Command Queue. If modifications have been made to the Command Queue and not saved, you are prompted to save the changes.

To change from one command queue to another, you must first close the current command queue, and then open another using the File > Command Queue menu selection.

## Glossary of Terms

| | |
|---|---|
| ***antecedent*** | The subject node in a Mask or Requires constraint |
| ***black-box view*** | A view of the system where the interface definition is physical (e.g., screens, files) but you do not see how the data was processed (i.e., the internals are logically defined in terms of what happens not how). See also white-box view. |
| ***cause*** | A qualified condition which leads to an effect (i.e., an input). |
| ***cause construct*** | In a compound relationship, a set of two or more causes which use the same relational operator (i.e., the causes listed between parenthesis in a compound Relations statement). |
| ***cause state*** | The true, false or indeterminate state of any given cause, as it exists in the functional variation being evaluated. |
| ***cause-effect graph*** | A notational convention for representing the relationships and conditions present in a requirements specification; this may be accomplished via declarative statements in a text file and/or pictorial graph representations. |
| ***compound relationship*** | Any Relations statement present in an input cause-effect graph file that uses two or more different relational operators. |
| ***connective*** | Synonym for relational operator and logical operator; AND, OR, NAND, NOR, XOR, XNOR and Negation. |
| ***consequent*** | The object node(s) in a Mask or Requires constraint. |
| ***constraint*** | A qualification placed upon cause-effect graph relationships in order to limit or preclude certain combinations due to input boundary conditions. |
| ***effect*** | The result of one or more qualified conditions or causes (i.e., an output). |
| ***effect state*** | The true, false or indeterminate state of any given effect, as it exists in the functional variation being evaluated. |
| ***explicit node*** | A node that has been defined using a Nodes statement and referenced in a Relations statement. |

| | |
|---|---|
| ***extrapolate*** | Sensitize an effect node state if the state of any one cause is sufficient to determine the effect state when taking into account the relational operator. |
| ***extrapolated state*** | The true or false state of a node was established without using the combinations of node states present in the set of functional variations generated (see extrapolate above). |
| ***false state*** | A condition that does not exist; see also true state. |
| ***forced observable*** | An effect whose true and false states are not normally observable which the user would like to treat as observable in order to include certain untestable variations in the suite of test cases generated. These identify where diagnostic probes need to be inserted into the software. |
| ***fully-sensitized variation*** | A statement of the sensitized (true, false or indeterminate) state of all nodes relevant to a functional variation. |
| ***functional specification*** | The document that defines, in user terminology, what the system should do. Aliases: requirements specification, external specification, logical specifications. |
| ***functional variation*** | One or more relationships consisting of one or more cause states and a resultant effect state, all of which are derived from a single Relations statement. The combination of cause states presented are the minimum combinations necessary to detect a fault during testing of the relationship. |
| ***implicit node*** | A node internally created and used by BenderRBT. |
| ***indeterminate*** | State of a node; third node state possible (i.e., true, false, indeterminate); node state established as result of the application of a Mask constraint; aliases: do not care, masked, irrelevant. |
| ***Infeasible*** | A functional variation that the user has precluded from being considered for inclusion in any test case due to the imposition of one or more constraints; a functional variation whose effect state is infeasible (i.e., illogical, not possible) after one or more of the relationship's causes have been constrained. |
| ***Intermediate effect*** | A node which is both a cause and an effect; i.e., it is both an effect of one or more causes and a cause of one or more effects. |
| ***local relationship*** | A cause-effect relationship that addresses only one effect. |

| | |
|---|---|
| ***logical operator*** | Synonym for relational operator and connective; i.e., AND, OR, NAND, NOR, XOR, XNOR and Negation. |
| ***masked node*** | A node that is an object of a Mask constraint and the mask subject node state currently exists. |
| ***node*** | A single entity within a requirements specification. Each defined node should clearly identify the variable and value for that variable or clearly define a system state. |
| ***object node*** | The second (and subsequent) node(s) named in a Mask or Requires constraint statement; see also consequent. |
| ***observable*** | An effect whose true and false states can be seen, detected and verified. |
| ***passive*** | The optional PAS node definition or subgraph designation; used to declare the node or subgraph as being outside of the desired scope of testing; used to differentiate the (normally) active (node or subgraph) portions of a graph from the inactive (i.e., passive) portions of a graph. |
| ***primary cause*** | A node that is not also an effect of one or more preceding causes; i.e., the beginning or entry-point node(s) in a cause-effect graph. |
| ***primary effect*** | A node that has no subsequent effects; i.e., the final or exit-point node(s) in a cause-effect graph. |
| ***reconvergent fanout*** | When the sensitized state of one or more causes results in the sensitizing of two or more logical paths through a graph which at some point converge at a common intermediate or primary effect. |
| ***relational operator*** | Synonym for connective and logical operator; AND, OR, NAND, NOR, XOR and Negation. |
| ***requirements specification*** | The document which defines, in user terminology, what the system should do. Aliases: functional specification, external specification, logical specifications. |
| ***sensitize*** | To establish the true, false or indeterminate state of a node. |
| ***sensitized state*** | The true, false or indeterminate state of a node was established by normal test case synthesis. |

| | |
|---|---|
| ***strong coverage*** | Within the context of the Coverage Analyzer Utility, strong coverage is tallied only for those functional variations where all of the variations derived from any given Relations statement are covered. See also weak coverage. |
| ***stuck-at-fault*** | The true or false state representation of a node does not change due to an error in the program under test's software logic. |
| ***subgraph*** | One or more cause-effect graph statements maintained in a file separate from (and referred to by) a main cause-effect graph file. |
| ***subject node*** | The first node named in a Mask or Requires constraint statement; see also antecedent. |
| ***synthesize*** | The combining of sensitized nodes into logical test case definitions. |
| ***true state*** | A condition which does exist; see also false state. |
| ***untestable*** | A functional variation whose effect state BenderRBT has been unable to sensitize due to observability issues; a functional variation whose effect state cannot be observed in both its true and its false state because the effect was not declared as, or is not observable (or forced-observable). |
| ***vector state*** | The true or false state of any given cause, as it was declared in the Relations statement. |
| ***weak coverage*** | Within the context of the Coverage Analyzer Utility, weak coverage denotes the simple percentage of any functional variations that have been covered by the selected (or completed) test cases. See also strong coverage. |
| ***white-box view*** | A view of the system where the interface definition is physical and how the transformation of data is accomplished is also defined (e.g., in terms of modules, physical tables). See also black-box. |